ILOG JViews 5.5

Graphics Framework User's Manual

December 2002

Printed in France

Table of Contents

About This Manual

ILOG JViews Graphics Framework is a foundation module in the ILOG JViews Component Suite. Completing the ILOG JViews Component Suite are:

- ILOG JViews Application Framework, providing an easy-to-use GUI editor for developing your graphical user interface (GUI) for applications based on the ILOG JViews Component Suite.
- ILOG JViews Graph Layout, providing special support for applications that need to display graphs (networks) of nodes and links.
- ILOG JViews Maps, allowing you to develop applications displaying maps using ILOG JViews.
- ILOG JViews Prototypes, allowing you to create custom graphic objects for specific application domains.
- ILOG JViews Gantt Chart, to manage the editing and the display of scheduling data.
- ILOG JViews Stylable Data Mapper (SDM), introduces the MVC architecture into standard ILOG JViews displays. It allows developers to build many ILOG JViews supervision and planning displays quickly, while maintaining full control over customizability. A specific version of SDM dedicated to building displays for Workflow applications is shipped with this package.

- ILOG JViews Charts, providing a data visualization library for displaying your data in charts, which can be easily integrated into any Swing-enabled Java application. The package includes such features as server-side charts with thin-client support, load-on-demand, external data connectors (XML, Swing TableModel, JDBC), JavaBeans, and integration with other ILOG JViews packages.

What Is in This Manual

This manual describes the graphics framework of ILOG JViews, a hierarchy of classes that let you create various high-level *graphic objects*. It also describes how to coordinate a large quantity of graphic objects through the use of a *manager*, that is, through the `IlvManager` class and its associated classes.

This manual contains the following chapters:

- Chapter 1, *Introducing ILOG JViews Graphics Framework*, provides an overview of the ILOG JViews Graphics Framework, explaining its importance in the ILOG JViews Component Suite.
- Chapter 2, *Getting Started with Graphics Framework*, presents a tutorial which demonstrates the basic concepts of ILOG JViews.
- Chapter 3, *Graphic Objects*, describes the hierarchy of classes that lets you create various high-level graphic objects.
- Chapter 4, *Managers*, describes how to coordinate a large quantity of graphic objects through the use of a manager.
- Chapter 5, *Graphers*, presents the high-level ILOG JViews grapher class allowing you to create graphic programs that both include and represent hierarchical information with graphic objects.
- Chapter 6, *Nested Managers and Nested Graphers*, describes how to add a manager in another manager or a grapher in another grapher.
- Chapter 7, *ILOG JViews Graphics Framework Printing*, describes the Graphics Framework extensions of the ILOG JViews print capabilities. The basic printing framework is described in a separate manual, the *ILOG JViews Print Framework User's Manual*.
- Chapter 8, *Composer, the ILOG JViews Editor*, describes how to use the ILOG JViews editor.
- Chapter 9, *Graphics Framework JavaBeans*, shows you how to create an application using ILOG JViews Beans.

- Chapter 10, *Thin-Client Support in ILOG JViews Graphics Framework*, describes ILOG JViews support for creating “light” applications on the client side backed by ILOG JViews functionality on the server side.
- Chapter 11, *Scalable Vector Graphics and ILOG JViews*, shows you how to use Scalable Vector Graphics (SVG) in your ILOG JViews applications.

The appendixes provide auxiliary and reference information as follows:

- Appendix A, *Differences Between ILOG Views and ILOG JViews*, describes the differences between the C++ and Java versions of the ILOG Views libraries.

At the end of the manual you will find a *Glossary* containing definitions of the basic technical terms used in this manual.

CHAPTER

1

Introducing ILOG JViews Graphics Framework

ILOG JViews Graphics Framework is a structured 2D graphics package for creating highly custom, visually rich graphical user interfaces. It complements the simple components provided by Swing or AWT, allowing Java GUI programmers to develop far more intuitive displays of information. Examples of such types of displays include schematics, workflow and process flow diagrams, command and control displays, network management displays, and any application requiring a map. The ILOG JViews Graphics Framework (or Graphics Framework, for short) is ideal for the rapid development of these custom GUIs.

You will find the following additional topics in this introduction:

- *The Graphics Framework Package*
- *Foundation of the ILOG JViews Component Suite*
- *A Basic Graphics Application Model*

The Graphics Framework Package

The ILOG JViews Graphics Framework package consists of a set of Java Beans, a Developer's API, and a graphical editor.

The JavaBean components allow you to get a fast start with the Graphics Framework. You can bring up your favorite Integrated Development Environment (IDE), import the Beans, connect them to any other Beans, compile, and run. You can have a working example of an ILOG JViews applet or application in just a few minutes. These Beans are an excellent beginning when learning the features of ILOG JViews, and can be customized for delivery to your end users as well.

Most applications will, however, require functionality that goes far beyond what these pre-packaged Beans offer, and this is why the Graphics Framework is delivered with a Developer's API. This API is a fully documented class library allowing you to build the custom look-and-feel that your application needs. Once you understand how the basic parts of the library work, you can customize or extend it. The library's architecture is completely open to extension.

Finally, the Graphics Framework is delivered with Composer, a full-featured Java application that can be used to draw and edit ILOG JViews graphic objects and save the results in an `.ivl` (**I**LOG JViews **V**ector **L**anguage) or SVG (Scalable Vector Graphic) file. Because Composer is delivered with a complete set of source code, many developers find it to be an excellent starting point for developing their own custom editors (which they then deliver to their end users).

Foundation of the ILOG JViews Component Suite

The Graphics Framework package provides the base functionality for the graphics application built with the ILOG JViews Component Suite. It handles the creation and manipulation of basic graphic objects such as lines, rectangles, and labels, as well as any of the custom objects that you might create. The Graphics Framework also provides the optimized data structures that allow the graphic objects to be panned, zoomed, and selected with optimal performance. Finally, it provides a set of behaviors that can be used to define the user interactions with the display and the graphic objects. All of the other modules in the ILOG JViews Component Suite rely on the Graphics Framework for these core-level services. This is illustrated in Figure 1.1.

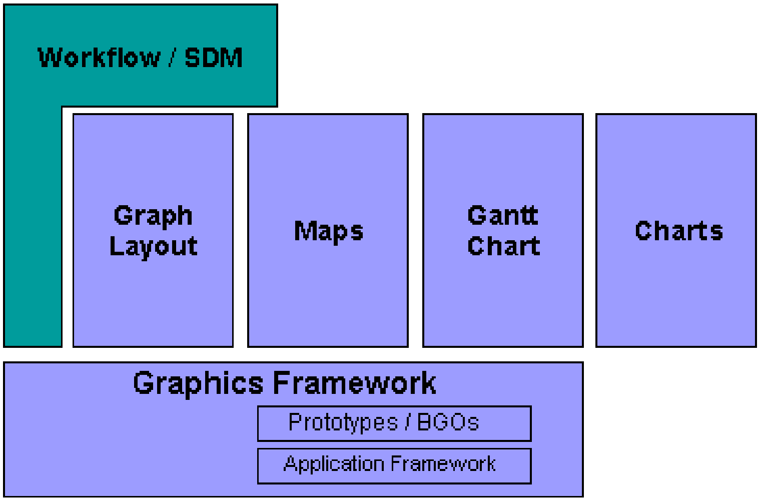


Figure 1.1 *ILOG JViews Suite*

A Basic Graphics Application Model

To use the ILOG JViews Component Suite effectively, you will need to understand how to use the ILOG JViews Graphics Framework module. And to use the Graphics Framework, you will need to understand the basic object model, that is, the core Java objects and their relationships to each other.

A basic graphics application needs just a few parts:

- Graphic objects (manipulated with zoom, resize, select, and draw functionality),
- A data structure to put them into, and
- A viewport (typically a rectangular area in a window on the display) to draw them into.

In the ILOG JViews world, we formally refer to these parts as `IlvGraphic`, `IlvManager`, and `IlvManagerView` objects, respectively. Their organization is shown in Figure 1.2.

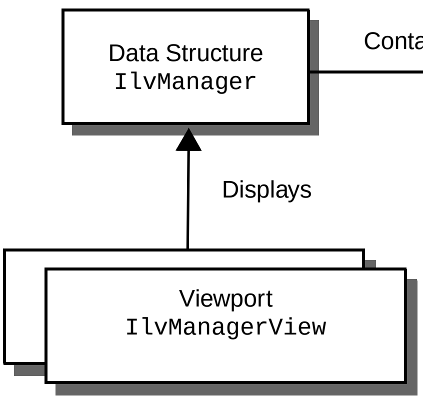


Figure 1.2 *Basic ILOG JViews Classes*

The basic ILOG JViews classes are described further in:

- *The Graphic Object: IlvGraphic*
- *The Data Structure: IlvManager*
- *The Viewport: IlvManagerView*

The Graphic Object: IlvGraphic

An `IlvGraphic` graphic object typically represents some custom entity in the end-user's application domain. For example, in a geographic display for a telecom network, there may be lines, labels, and polygons that form the background map and some other more sophisticated objects that represent the telecom devices on the network.

The graphic framework comes with a large set of predefined graphic objects, such as rectangles, polylines, polygon, splines, and labels. Other domain-specific objects (such as the network devices in the above example) can be created by subclassing one of these objects or the base class object, `IlvGraphic`, or by grouping predefined objects together.

The Data Structure: IlvManager

The `IlvManager` data structure contains the graphic objects, therefore, it is the most important object of the Graphics Framework. The manager organizes graphic objects in several layers; graphic objects contained in a higher level layer are displayed in front of objects located in a lower layer.

The manager also provides the means to select the graphic objects. The library comes with several predefined ways to display a selected graphic object but you can create your own user-defined way.

The Viewport: IlvManagerView

An `IlvManagerView` viewport is designed to visualize the graphic objects of a manager. The class `IlvManagerView` is a component (subclass of the `java.awt.Component`) that you use in your AWT or Swing application to visualize the manager. A manager view lets you define which layer of the manager is visible for a view. In addition, you may use several manager views to visualize different areas of the manager. Of course, you may zoom and pan the content of the view.

CHAPTER

2

Getting Started with Graphics Framework

This chapter provides a tutorial explaining how to create a simple application using ILOG JViews Graphics Framework. It consists of four steps, indicated below. Through the construction of this application, we will briefly explain the following main concepts of ILOG JViews:

In *Step 1 - The Manager* on page 8:

- Creating the manager.
- Loading a file containing graphic objects into this manager.
- Creating the view to display the contents of the manager.

In *Step 2 - View Interaction* on page 11:

- Adding interaction to a view.

In *Step 3 - Using Events* on page 12:

- Listening to events sent by the manager view.

In *Step 4 - Manipulating Graphic Objects* on page 16:

- Adding and moving graphic objects in the manager.

Four sample Java source files are provided representing the four steps to the tutorial. The sample files are as follows: `Sample1.java`, `Sample2.java`, `Sample3.java`, and `Sample4.java`.

Running the Example

To run a sample file, do as follows:

We may begin with Sample1.java:

```
<installdir>/doc/usermansrc/graphicsuser/getstart/Sample1.java.
```

1. Go to the getstart directory at the above path.
2. Set the CLASSPATH variable to the ILOG JViews library: jviewsall.jar and to the current directory. On a Windows machine this will be:

   ```
   set CLASSPATH=.;<installdir>/classes/jviewsall.jar
   ```

3. Compile the Sample1.java file:

   ```
   javac Sample1.java
   ```

4. Run the application:

   ```
   java Sample1
   ```

Step 1 - The Manager

This part of the tutorial shows how to create a manager and its view, and how to load a file containing graphic objects into the manager. The manager organizes sets of graphic objects into multiple views and layers and provides the possibility of higher-level interactions. These features are brought out as the tutorial progresses. When running the Sample1 example, the first step to this tutorial, you see a scrolling window displaying a map of the United States.

Explanations of the Sample1.java file follow.

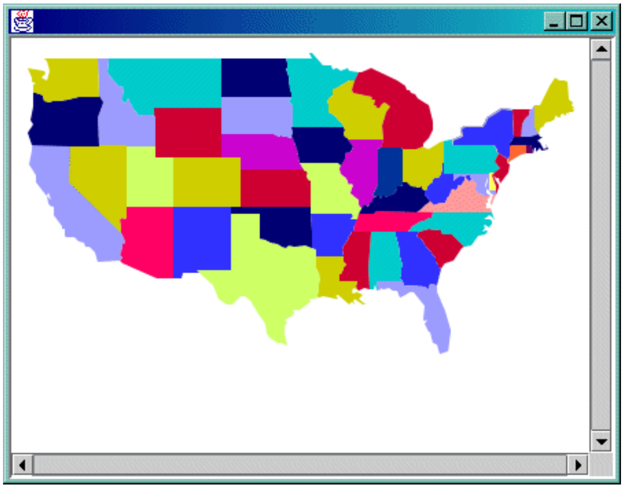


Figure 2.1 *Sample1*

Importing the Library and Packages

In the `Sample1.java` file, we first import the main ILOG JViews package as well as the ILOG JViews Swing package for the GUI components.

```
import ilog.views.*;
import ilog.views.swing.*;
```

Since we use AWT and Swing classes, we must import the `swing` and `awt` packages:

```
import javax.swing.*;
import java.awt.*;
```

The following sections explain the code extracted from `Sample1.java`.

Creating the Frame Class

You are now able to create the class named `Sample1`. This class has two fields, `manager` (class `IlvManager`) to store the graphic objects, and `mgrview`, (class `IlvManagerView`), to display the contents of the manager.

```
public class Sample1 extends JFrame
{
  IlvManager manager;
  IlvManagerView mgrview;
  ....
}
```

Creating the Manager

In the constructor, we create the manager:

```
...
  manager = new IlvManager();
   ...
}
```

Loading/Reading a File

Once the manager is created, we can read the `usa.ivl` file which can be found in the `getstart` directory. ILOG JViews Graphics Framework provides facilities to save and read graphic objects in a manager. These files are in the IVL format.

We need to catch the exception that may occur when reading the file. The method `read` of the class `IlvManager` may throw the following exceptions:

- `IOException` for basic IO errors.
- `IlvReadFileException`, if the format of the file is not correct (the file is not an `.ivl` formatted file) or if a graphic class needed to load the file cannot be found.

```
try {
  manager.read(new URL("usa.ivl");
} catch (Exception e) {}
```

Creating the View

Now we create a manager view to display the contents of the manager. A manager view is an instance of the `IlvManagerView` class. To associate it with the manager, all you have to do is provide the `manager` parameter as shown below.

> ***Note:*** *In this example, we use the class* `IlvJScrollManagerView`*. This class encapsulates the class* `IlvManagerView` *and provides scroll bars.*

```
mgrview = new IlvManagerView(manager);
getContentPane().setLayout(new BorderLayout(0,0));
getContentPane().add(new IlvJScrollManagerView(mgrview), BorderLayout.CENTER);
```

Testing the Application

To test the application, you need the `jviewsall.jar` JAR file in your classpath.

Step 2 - View Interaction

The second part of the tutorial, the `Sample2.java` file, is an extension of the `Sample1` file. Compile the `Sample2.java` file and run it as you did for `Sample1`. See *Running the Example* on page 8.

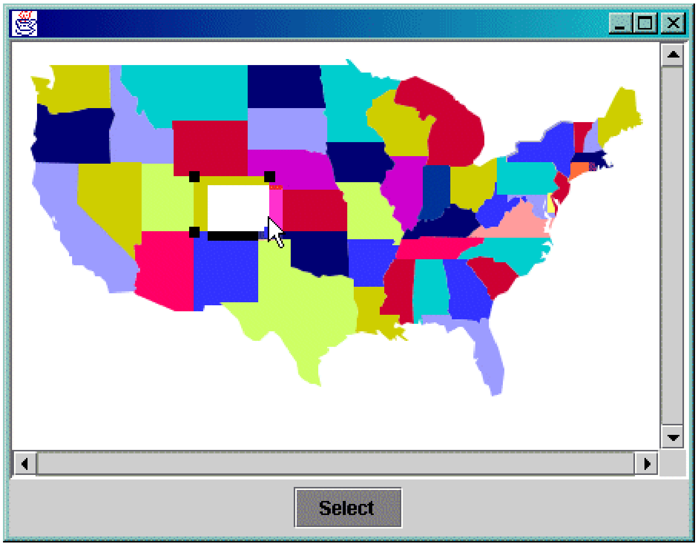


Figure 2.2 *Sample2 Running*

In this part of the tutorial, we add interaction to the view by placing a selection interactor on it. For this, we add a Select button and associate it with the interactor. By clicking on the Select button, the selection interactor is placed on the view. Once you click the button, you can select the graphic objects in the view (in our case the states of the United States), move them around, and modify their shape.

A selection interactor is an instance of the class `IlvSelectInteractor`, a subclass of the `IlvManagerViewInteractor` class. This *view interactor* will process all the input events, such as mouse and keyboard events, occurring in a manager view.

Adding the selectInteractor Field

To be able to use the class `IlvSelectInteractor`,, we must first import the ILOG JViews packages that contain the interactors:

```
import ilog.views.interactor.*;
```

then, we add a field named `selectInteractor` in class `Sample2`:

```
public class Sample2 extends JFrame
{
  IlvManager manager;
  IlvManagerView mgrview;
  IlvSelectInteractor selectInteractor;
  ....
}
```

Creating the Select Button

The following code creates a Select button and associates it with the `selectInteractor`:

```
void createButtons()
{
  JButton button;
  button = new JButton("Select");
  button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {
      if (selectInteractor == null)
         selectInteractor = new IlvSelectInteractor();
      if (mgrview.getInteractor() != selectInteractor)
         mgrview.setInteractor(selectInteractor);
    }
  });
  getContentPane().add(button, BorderLayout.SOUTH);
 }
```

When you click the Select button, the `actionPerformed` method first creates its interactor (if it has not already been done). Then, it installs the interactor on this view using the `setInteractor` method. Once the interactor is installed, you can select, move, and modify the graphics objects displayed in the view.

Step 3 - Using Events

The third part of the tutorial, the `Sample3.java` file, is an extension of the `Sample2` file. Compile the `Sample3.java` file and run it as you did for the previous sample files. See *Running the Example* on page 8.

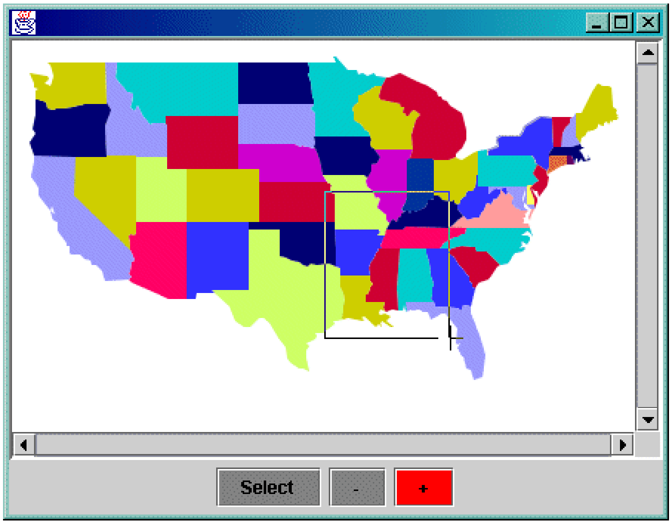


Figure 2.3 *Sample3 Running*

In Step 3, we show the use of events delivered by the view. To demonstrate this we will use the `InteractorListener` interface which will allow us to listen for a change of interactors. There are three buttons in the example, each with an associated interactor. Clicking one button and then another changes the ‘engaged’ interactor accordingly.

Two new interactors are placed on the view: `IlvZoomViewInteractor` and the `IlvUnZoomViewInteractor`. These interactors allow you to drag a rectangle on the view to zoom in and out on this area. The third interactor is the `IlvSelectInteractor` (of `Sample2`). Their respective buttons are created inside a Swing JPanel, which automatically aligns them as seen in the above illustration.

Adding New Interactor Fields

To accomplish our task, we add the `zoomInteractor` and `unzoomInteractor` fields along with the necessary interactor button fields to the `Sample3` applet class:

```
public class Sample3 extends Applet
{
  IlvManager manager;
  IlvManagerView mgrview;
  IlvSelectInteractor selectInteractor;
  IlvManagerViewInteractor zoomInteractor, unzoomInteractor;
  Button selectButton, zoomButton, unZoomButton;
  ....
}
```

Creating the Interactor Buttons

The `createInteractorButtons` method will create three buttons (Select, -, and +) that will be stored in the `selectButton`, `zoomButton`, and `unZoomButton` fields of the object.

```
void createInteractorButtons() {
  Panel buttons = new Panel();
  selectButton = new Button("Select");
  selectButton.setBackground(Color.gray);
  selectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      if (selectInteractor == null)
        selectInteractor = new IlvSelectInteractor();
        if (mgrview.getInteractor() != selectInteractor)
          mgrview.setInteractor(selectInteractor);
        }
    }
  });

  buttons.add(selectButton);
  unZoomButton = new Button("-");
  unZoomButton.setBackground(Color.gray);
  unZoomButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      if (unZoomInteractor == null)
        unZoomInteractor = new IlvUnZoomViewInteractor();
      if (mgrview.getInteractor() != unZoomInteractor)
        mgrview.setInteractor(unZoomInteractor);
    }
  });

zoomButton = new Button("+");
  zoomButton.setBackground(Color.gray);
  zoomButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      if (zoomInteractor == null)
        zoomInteractor = new IlvZoomViewInteractor();
      if (mgrview.getInteractor() != zoomInteractor)
       mgrview.setInteractor(zoomInteractor);
    }
  });
  buttons.add(zoomButton);
  getContentPane().add(buttons, BorderLayout.SOUTH);
}
```

As we now have three possible interactors, the action performed when clicking on a button removes the previously installed interactor and installs the new one by calling the `setInteractor` method of the `IlvManagerView` class.

Listening for a Change of Interactors

In the Sample3.java file, you will notice that the class implements the interface InteractorListener. You may also have noticed that we have imported a new package, ilog.views.event, which is the package that contains the ILOG JViews event classes. The InteractorListener interface includes one method:

```
void interactorChanged(InteractorChangedEvent event)
```

In this example, we want the selected button to be red when its corresponding interactor is attached to the view.

The interactorChanged method will be called when the interactor changes on the view (as soon as the object, the instance of Sample3, is registered as a listener; see *Registering the Listener* on page 16). The parameter is an event that contains the old and the new interactor. We simply change the background color of the button corresponding to the newly installed interactor to red:

```
public void interactorChanged(InteractorChangedEvent event)
{
  IlvManagerViewInteractor oldI = event.getOldValue();
  IlvManagerViewInteractor newI = event.getNewValue();
  if (oldI == selectInteractor)
    selectButton.setBackground(Color.gray);
  else if (oldI == zoomInteractor)
    zoomButton.setBackground(Color.gray);
  else if (oldI == unZoomInteractor)
    unZoomButton.setBackground(Color.gray);
  if (newI == selectInteractor)
    selectButton.setBackground(Color.red);
  else if (newI == zoomInteractor)
    zoomButton.setBackground(Color.red);
  else if (newI == unZoomInteractor)
    unZoomButton.setBackground(Color.red);
}
```

Registering the Listener

It is not enough to implement the interface. We must not forget to register this new listener with the view. This is done by calling the `addInteractorListener` method in the `init` method:

```
...
    manager = new IlvManager();
    try {
      manager.read("usa.ivl");
    } catch (Exception e) {}
    mgrview = new IlvManagerView(manager);
    setLayout(new BorderLayout(0,0));
    getContentPane().add(new IlvJScrollManagerView(mgrview),
                         BorderLayout.CENTER);
    createButtons();
    mgrview.addInteractorListener(this);
 ...
```

Step 4 - Manipulating Graphic Objects

The fourth part of the tutorial, the `Sample4.java` file, is an extension of the `Sample3` file. Compile the `Sample4.java` file and run it as you did for the previous sample files. See *Running the Example* on page 8.

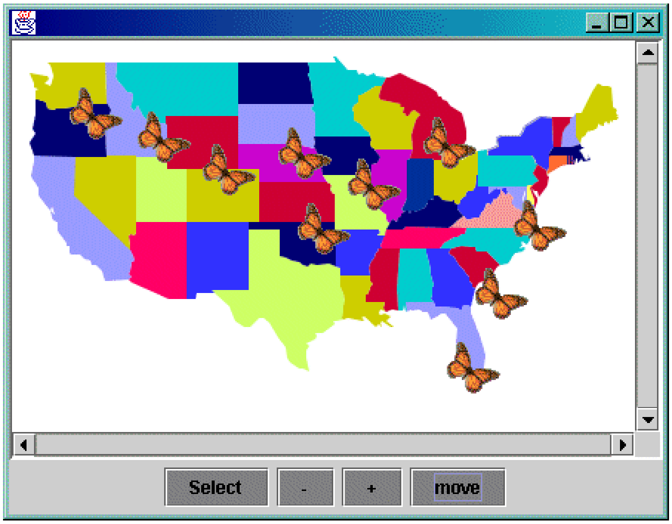


Figure 2.4 *Sample4 Running*

This part of the tutorial concerns the manipulation of graphic objects, demonstrating how to create and add graphic objects to the manager and how to change their location.

Adding Graphic Objects

To be able to manipulate graphic objects, we must first import the ILOG JViews package that contains the graphic objects:

```
import ilog.views.graphic.*;
```

In this example, we implement the `addObjects` method which adds ten objects of the `IlvIcon` class to the manager:

```
public void addObjects()
{
  manager.setSelectable(0, false);

    for (int i = 0 ; i < 10 ; i++) {
        IlvGraphic obj = new IlvIcon("image.gif", new IlvRect(0,0,37,38));
       manager.addObject(obj, 1, false);
    }
}
```

The first line in this method calls the `setSelectable` method on the manager with `0` and `false` as its parameters:

```
manager.setSelectable(0, false);
```

The first parameter, `0`, specifies the layer in the manager to which the method applies. The second parameter, `false`, specifies whether objects in the layer passed as the first parameter can be selected (`true`) or not (`false`).

Objects in a manager can be stored in different layers, which are identified by indices. Layers are drawn on top of each other, starting at index 0. In other words, the first layer is assigned the index 0, the second layer, the index 1, and so on, with the objects stored in a higher screen layer being displayed in front of objects in lower layers.

In the `usa.ivl` file loaded in the manager, the objects that make up the map are stored in layer `0`. Calling the `setSelectable` method with `0` and `false` as parameters specifies that the map (layer `0`) cannot be selected, and hence, cannot be modified.

The following `addObject` method adds the `IlvIcon` objects to layer 1 of the manager:

```
manager.addObject(obj, 1, false);
```

> ***Note:*** *T he `false` parameter of this method specifies that the redraw is not to be triggered. Here no redraw is needed because the application is not visible when this code is executed.*

Test the interface of the application by clicking on the objects with the mouse. You can see that the new objects are selectable, whereas you can no longer select or modify the map.

Moving Graphic Objects

In `Sample4`, we have added a new button to move the `IlvIcon` objects in a random way:

```
 Button moveButton = new Button("move");
    moveButton.setBackground(Color.gray);
    moveButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        moveObjects();
      }
    });
    buttons.add(moveButton);
  }
```

The movement of the objects is implemented in the `moveObjects` method of the class `IlvGraphic`. This method gets the objects of layer 1 (our new objects), and, for each of these objects, finds a random object in layer 0 and moves these objects of layer 1 to the center of those objects of layer 0.

```
void moveObjects() {
  IlvGraphic state=null, obj=null;
  // get objects in layer 1
  IlvGraphicEnumeration objects, states;
  for (objects = manager.getObjects(1); objects.hasMoreElements();) {
    obj = objects.nextElement();
    // get an random object in layer 0
    states = manager.getObjects(0);
    int index = (int)((double)manager.getCardinal(0)*Math.random());
    state = states.nextElement();
    for (int i = 1 ; i < index; i++)
      state = states.nextElement();
    if (state != null) {
      // move the object.
      IlvRect bbox = state.boundingBox(null);
      manager.moveObject(obj, bbox.x+bbox.width/2,
                              bbox.y+bbox.height/2, true);
    }
  }
}
```

CHAPTER

3

Graphic Objects

This section describes the Graphics Framework hierarchy of classes which lets you create various high-level graphic objects. In this section, we describe the starting point for these objects, the class `IlvGraphic`. The topics covered are:

Introducing Graphic Objects

A *graphic object* is an object that users can view on their screen. When you display a graphic object, you associate its coordinates with the coordinate system of a particular *graphic bag*. A graphic bag is an interface that describes the methods to be implemented by a class that contains several graphic objects. An example of a graphic bag is the class `IlvManager`, which can manage a large number of graphic objects. For more information see Chapter 4, *Managers*.

Every graphic object has an x value, a y value, and dimensions (that is, width and height). The x and y values define the upper-left corner of the graphic object's *bounding box*, which is the smallest rectangle containing the entire area of the object. You define the exact shapes of graphic objects in your ILOG JViews-based programs and then build them using various drawing methods. Other methods provide you with information about your graphic objects and let you carry out geometric tests concerning the shapes that you are using. For example, you can check whether or not a point with given coordinates lies inside a certain object form.

The Class IlvGraphic

ILOG JViews graphic objects inherit attributes from the `IlvGraphic` abstract base class. This class allows an ILOG JViews graphic object to draw itself at a given destination port. If required, the coordinates of the graphic object may also be transformed by an object associated with the `IlvTransformer` class. The class `IlvGraphic` has methods that allow you to set and change geometric dimensions but does not actually implement these methods. They are declared as nonfinal methods and are defined to perform various operations in the classes that inherit `IlvGraphic` attributes. Although the methods to manipulate geometric shapes and graphic attributes exist, their implementations are empty. Several methods are given to set and get user properties that can be associated with an object for application-specific purposes.

Hierarchy of ILOG JViews

ILOG JViews Graphics Framework provides a wide range of predefined graphic objects to create a sophisticated application with minimum coding. These objects/classes are illustrated in the following diagram.

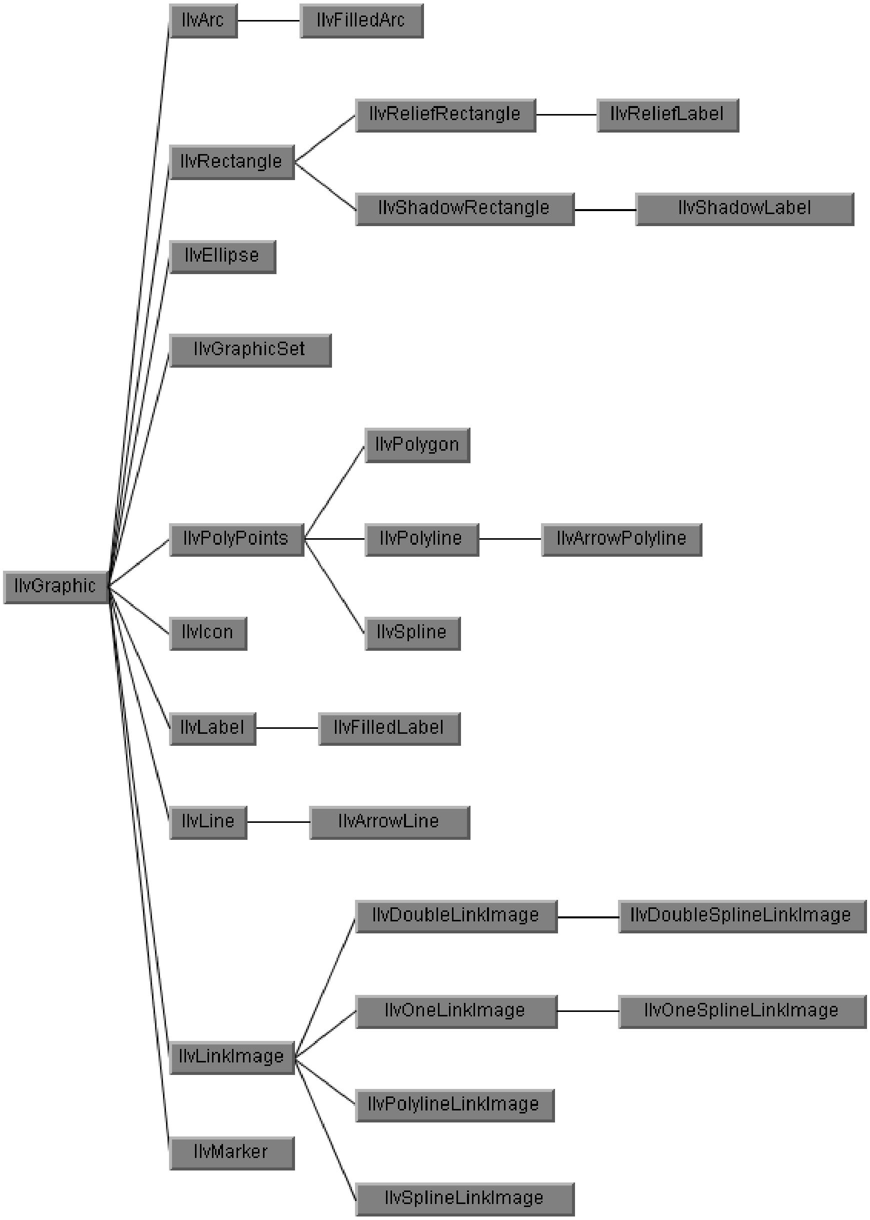


Figure 3.1 *Partial Class Hierarchy of the ILOG JViews Graphic Objects*

Geometric Properties

A graphic object is defined by a set of geometric properties, such as its location, size, shape, the way in which it is drawn, and so on. These properties are defined by a special set of methods. Of these methods, `draw` and `boundingBox` are fundamental and should be defined jointly. They are of the `IlvGraphic` class and are covered in the following sections.

The boundingBox method

The bounding box defines the smallest rectangle encompassing the graphic object. It is returned by the following method:

```
public IlvRect boundingBox(IlvTransformer t)
```

The `IlvTransformer` parameter is the 2D transformation matrix used to draw the object in a particular drawing port (see *transformer*). This transformation may correspond to a zoom, a rotation, or a translation of the graphic object in the destination drawing port. The method must then return the rectangle that contains the graphic object when it is drawn using the specified transformation.

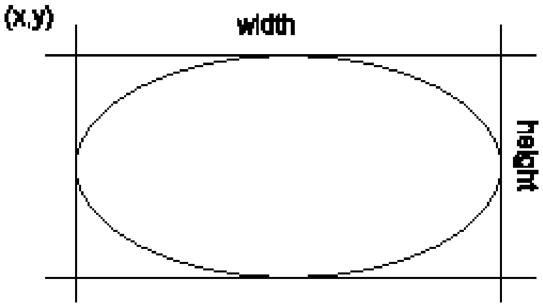


Figure 3.2 *The Bounding Box of a Graphic Object*

The following example defines the shape of a graphic object with the `drawrect` field. In order to return the bounding box of the object, the `boundingBox` method simply applies the transformer to the rectangle:

```
class MyRectangle extends IlvGraphic
{
   // The geometric rectangle that defines the object.
   final IlvRect drawrect = new IlvRect();

   //constructor
   public MyRectangle(float x, float y, float width, float height)
   {
    drawrect.reshape(x, y, width, height);
   }

   // The bounding box method.
   public IlvRect boundingBox(IlvTransformer t)
   {
      //Copies the original rectangle to avoid its modification
      IlvRect rect = new IlvRect(drawrect);
      if (t != null)
        t.apply(rect);
      return rect;
   }
}
```

The method `boundingBox` is a very important method. Since it is called very frequently, it must be written in a highly optimized way.

The draw Method

The `draw` method is used to draw the graphic object. The signature of the method is the following:

```
public void draw(Graphics dst, IlvTransformer t)
```

The `dst` parameter is the destination `Graphics` where the object is drawn. As in the `boundingBox` method, the `IlvTransformer` parameter is the 2D transformation matrix used to draw the object in the drawing port.

> ***Note:*** *Everything that is drawn with this method must be drawn inside the bounding rectangle of the object (the bounding rectangle of the object being the result of the call to the method* `boundingBox` *with the same transformation parameter). This is why these two methods should be defined jointly.*

In order to draw the object, you will use the drawing methods of the AWT `Graphics` class. If you use Java 2 and need to perform Java2D drawings, you can cast the `dst` parameter in a `Graphics2D` object and then use the drawing methods of this class.

Zoomable and Nonzoomable Objects

A graphic object is said to be *zoomable* if its bounding box follows the zoom level. In other words, the result of calling the method `boundingBox` with a *transformer* is the same as when calling `boundingBox` with a `null` transformer and then applying the transformer to the resulting rectangle. That is:

obj.**boundingBox**(*t*) = *t*.**apply**(*obj*.boundingBox(null))

Zoomable and nonzoomable objects are managed in very different ways in ILOG JViews, zoomable objects being managed in a more optimized way. To know whether an object is zoomable, call the `zoomable` method:

```
public boolean zoomable()
```

The returned value for the class `IlvGraphic` is `true`.

Testing Whether a Point is Part of an Object Shape

The method `contains` is called by interactors to check whether a point is part of an object shape.

```
public boolean contains(IlvPoint p, IlvPoint tp, IlvTransformer t)
```

The default implementation of this method simply checks whether the specified point lies inside the bounding rectangle of the object. You may override this method so that it returns `false` for the transparent area of your object.

Moving and Resizing a Graphic Object

The class `IlvGraphic` provides many methods for moving and resizing a graphic object:

- `public void move(float x, float y)`

  Moves the upper-left corner of the bounding rectangle of the object to (x,y).

- `public void move(IlvPoint p)`

  Moves the upper-left corner of the bounding rectangle of the object to the point `p`.

- `public void moveResize(IlvRect size)`

  Sets the bounding rectangle of the object to the `IlvRect` parameter.

- `public void translate(float dx, float dy)`

  Translates the bounding rectangle of the object by the vector (`dx`, `dy`).

- `public void rotate(IlvPoint center, double angle)`

  Rotates the object around the point `center` by an angle of `angle` degrees.

- `public void scale(double scalex, double scaley)`

  Resizes the bounding rectangle of the object by a factor (`scalex`, `scalex`).

- `public void resize(float neww, float newh)`

  Modifies the bounding rectangle of the object with the new size (`neww`, `newh`).

All the above methods call `applyTransform` to modify the bounding rectangle of the graphic object.

```
public void applyTransform(IlvTransformer t)
```

This is the only method that needs to be overridden in order to handle the transformation of an object correctly. The following code example shows how the `applyTransform` method may be used in our example class, `MyRectangle`:

```
class MyRectangle extends IlvGraphic
{
   // The rectangle that defines the object.
   final IlvRect drawrect = new IlvRect();

   ...

   public void applyTransform(IlvTransformer t)
   {
      t.apply(drawrect);
   }
}
```

The method simply applies the transformation to the rectangle.

> ***Note:*** *Graphic objects stored in a manager (class* `IlvManager` *and its subclasses) are located in a quadtree. This means that you cannot simply call* `move` *on a graphic object because the quadtree must be notified of the modification of the graphic object. Every method that modifies the bounding rectangle of the object must use the method* `applyToObject` *of the class* `IlvManager`*. This method applies a function to an object and notifies the quadtree of the modification to the bounding rectangle. The class* `IlvManager` *also includes several convenient methods to move and reshape a graphic object managed by this manager. They are shown below:*

```
public void moveObject(IlvGraphic, float, float, boolean)

public void reshapeObject(IlvGraphic, IlvRect, boolean)
```

User Properties of Graphic Objects

A set of user properties can be associated with graphic objects. User properties are a set of key-value pairs, where the key is a `String` object and the value may be any kind of information value. These user property methods of the class `IlvGraphic` let you easily connect information that comes from your application to your graphic objects. You can keep track of the graphic part of your application by storing the references to objects you create and connecting this graphic part to the application by means of user properties, as in the following example:

```
Integer index = new Integer(10);
String key = "objectIndex";
myobject.setProperty(key, index);
```

The following `IlvGraphic` methods help you manage the properties of an object:

```
public boolean hasProperty(String key, Object value)

public boolean removeProperty(String key)

public Object getProperty(String key)

public boolean replaceProperty(String key, Object value)

public void setProperty(String key, Object value)
```

Input/Output Operations

The ILOG JViews library provides the following two classes for saving graphic objects to, and loading graphic objects from, a stream:

- `IlvOutputStream`, used by the class `IlvManager` to save all the graphic objects that it contains
- `IlvInputStream`, allowing a file generated by `IlvOutputStream` to be read into an `IlvManager`

Graphic objects can always be written into an `IlvOutputStream` because they inherit the `write` method of the `IlvGraphic` class:

```
public void write(IlvOutputStream stream) throws IOException
```

To save the information contained in your class, you can override this method and use the methods of the class `IlvOutputStream`. When overriding this method, you must not forget to call the `write` method of the superclass to save the information related to the superclass. You will obtain something that resembles the following example:

```
public void write(IlvOutputStream stream) throws IOException
{
  // write fields of super class
  super.write(stream);
  // write fields of my class
  stream.write("color", getColor());
  stream.write("thickness", getThickness());
  ....
}
```

To read your graphic object from an `IlvInputStream`, you must create a constructor with an `IlvInputStream`. This constructor is mandatory even if you have not overridden the `write` method. The corresponding constructor in the class `IlvGraphic` is:

```
public IlvGraphic(IlvInputStream stream) throws
IlvReadFileException
```

Assuming that `MyClass` is the name of your class, your new constructor will look like this:

```
public MyClass(IlvInputStream stream) throws IlvReadFileException
```

In the body of this constructor, you first call the corresponding constructor in the superclass, then you read the information you have saved in the `write` method. In the above example, the corresponding constructor is:

```
public MyClass(IlvInputStream stream) throws IlvReadFileException
{
  super(stream);
  setColor(stream.readColor("color"));
  setThickness(stream.readInt("thickness"));
  ...
}
```

The Graphic Bag

Graphic objects are placed in a *graphic bag*. A graphic bag (interface `IlvGraphicBag`) is an object that contains several graphic objects, which can be added or removed. The interface `IlvGraphicBag` is implemented by the class `IlvManager`. This class allows you to manage a large set of graphic objects. The following method returns the graphic bag, if there is one, where the object is located:

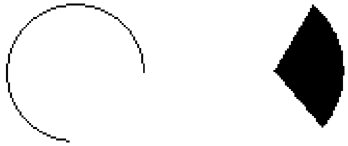
```
public IlvGraphicBag getGraphicBag()
```

Read Chapter 4, *Managers* for more information on this topic.

Predefined Graphic Objects

This section describes basic classes that provide you with ready-to-use drawing objects.

- **`IlvArc`**



An `IlvArc` object appears as an outlined, a filled, or filled and outlined arc of an ellipse.

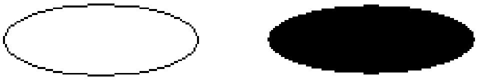
- **`IlvComponentGraphic`**

An `IlvComponentGraphic` object is a wrapper class that lets you embed a lightweight component in a manager.

- **`IlvJComponentGraphic`**

An `IlvJComponentGraphic` object is a wrapper class that lets you embed a Swing `JComponent` in a manager.

- **`IlvEllipse`**



An `IlvEllipse` object appears as an outlined, a filled, or filled and outlined ellipse.

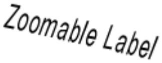
- **`IlvIcon`**

An `IlvIcon` object appears as an image.

- **`IlvLabel`**

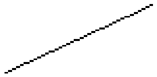
An `IlvLabel` object appears as a single line of text. It cannot be zoomed in or reshaped.

- **`IlvZoomableLabel`**



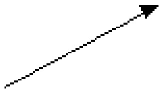
In Java 2, an `IlvZoomableLabel` object appears as a single line of text. As you can see from its image, it can be zoomed, reshaped and rotated.

- **`IlvLine`**



An `IlvLine` object appears as a straight line between two given points.

- **`IlvArrowLine`**

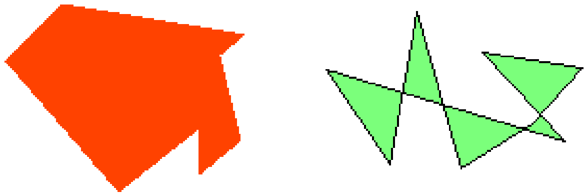


An `IlvArrowLine` object appears as a straight line between two given points, with a small arrowhead drawn at the end of the trajectory.

- **`IlvPolyPoints`**

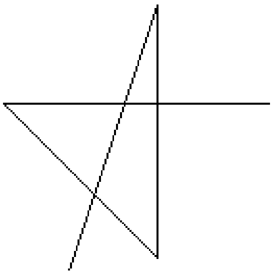
`IlvPolyPoints` is an abstract class from which every class having shapes made up of several point coordinates is derived.

- **`IlvPolygon`**



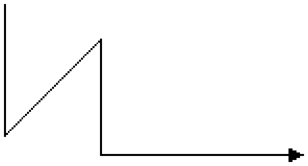
An `IlvPolygon` object appears as a filled, outlined, or filled and outlined polygon.

- **`IlvPolyline`**



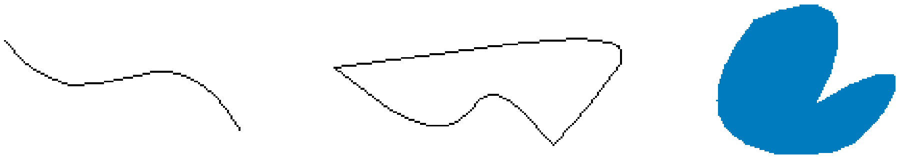
An `IlvPolyline` object appears as connected segments.

- **`IlvArrowPolyline`**



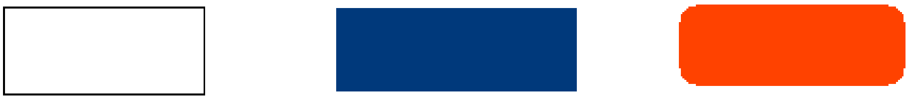
An `IlvArrowPolyline` object appears as a polyline and adds one or more arrows to the various lines.

- **`IlvSpline`**



An `IlvSpline` object appears as a Bézier spline. It can be either opened or closed, and may also be filled.

- **`IlvRectangle`**



An `IlvRectangle` object appears as a closed rectangle. It can be outlined, filled, or filled and outlined. You can also set rounded corners on the `IlvRectangle`.

- **`IlvReliefRectangle`**

An `IlvReliefRectangle` object appears as a filled rectangle in relief.

- **`IlvReliefLabel`**

An `IlvReliefLabel` object appears as a relief rectangle holding a single line of text.

- **`IlvShadowRectangle`**

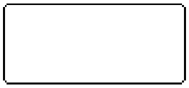
An `IlvShadowRectangle` object appears as a rectangle with a shadow underneath.

- **`IlvShadowLabel`**

Shadow label

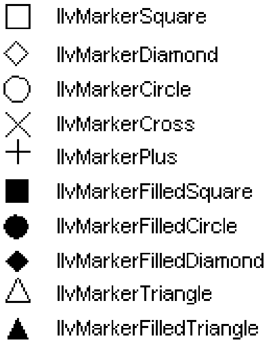
An `IlvShadowLabel` object appears as an `IlvShadowRectangle` with a label.

- **`IlvRoundRectangle`**



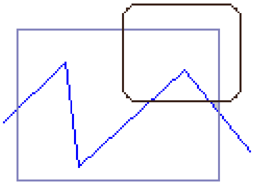
An `IlvRoundRectangle` object appears as a closed, round-cornered rectangle.

- **`IlvMarker`**



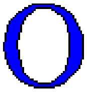
An `IlvMarker` object is a nonzoomable object that displays a graphic symbol.

- **`IlvGraphicSet`**



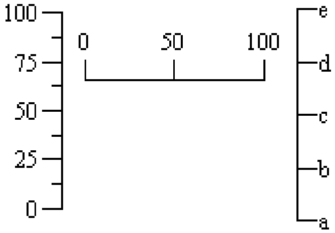
An `IlvGraphicSet` object is an object that groups graphic objects together.

- **`IlvGraphicPath`**



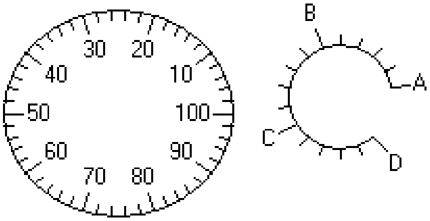
An `IlvGraphicPath` object is a set of polypoints that can be drawn as polylines or as polygons. Depending on the position of its points, a polypoint may either appear as an ordinary polygon or as a hole in another polygon.

- **`IlvRectangularScale`**



An `IlvRectangularScale` object displays a vertical or horizontal scale.

- **`IlvCircularScale`**



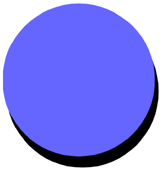
An `IlvCircularScale` object displays a circular scale defined by a portion of an ellipse, a starting angle and an angle range.

- **`IlvGeneralPath`**

An `IlvGeneralPath` can display any Java 2D `Shape` object. This means that it can represent curves, rectangles, ellipses, general paths, etc., and any combination of them. You can define Java 2D properties for these objects, such as `Paint` or `Stroke`. The last two objects with the fade-out effect are “gradiant paint” general path objects.

Creating a New Graphic Object Class

This section explains how to create a new graphic object class, `IlvShadowEllipse`, which inherits from `IlvGraphic`. The `IlvShadowEllipse` object is an ellipse object with a drop shadow, as seen below:



You can design such an object from scratch by implementing a subclass of the `IlvGraphic` class. In the following subsections you learn the most commonly used procedure. We show you how to define methods that deal with geometric properties and drawing, and how to make this object persistent. Click here to see the complete source code of the example.

Basic Steps

The procedure to create a derived class of the `IlvGraphic` class can be broken down into three main steps:

1. Creating the class
2. Defining the constructor
3. Defining accessors

In this example we are going to create the class `ShadowEllipse`. This file is located at:

`<install>/doc/usermansrc/graphicsuser/ShadowEllipse.java`

Step 1 - Creating the Class

To create the class:

1. Create a file named `ShadowEllipse.java` that defines the new class and the necessary overloaded methods. Not every method needs to be overloaded.
2. Add the following statements at the beginning of your file:

```
import ilog.views.*;
import ilog.views.io.*;
import ilog.views.graphic.*;
```

These statements allow you to use the basic, the input/output, and the graphic packages of the ILOG JViews library.

3. Define a class that inherits from the `IlvGraphic` class.

   This class has two colors: one for the ellipse and one for the shadow. It also defines the thickness of the shadow.

4. Define the bounding rectangle of the object.

   For this, you add a member variable named `drawrect` of type `IlvRect`.

```
import ilog.views.*;
import ilog.views.io.*;
import ilog.views.graphic.*;
import java.awt.*;
import java.io.*;

/**
 * A shadow ellipse object. A graphic object defined by two
 * ellipses: The main ellipse and a second ellipse of the same
 * size underneath the first one that represents a shadow.
 */
public class ShadowEllipse
extends IlvGraphic
{
  /**
   * The definition rectangle of the ellipse.
   * This rectangle is the bounding rectangle of the
   * graphic object.
   */
  protected final IlvRect drawrect = new IlvRect();
  /**
   * The color of the ellipse.
   */
  private Color  color = Color.blue;
  /**
   * The color of the shadow.
   */
  private Color  shadowColor = Color.black;
  /**
   * The thickness of the shadow.
   */
  private int thickness = 5;
```

Step 2 - Defining the Constructor

A constructor is defined to create a new shadow ellipse. This constructor simply sets the value of the definition rectangle:

```
/**
 * Creates a new shadow ellipse.
 * @param rect the bounding rectangle of the shadow ellipse.
 */
public ShadowEllipse(IlvRect rect)
{
  super();
  // Stores the bounding rectangle of the object.
  drawrect.reshape(rect.x, rect.y, rect.width, rect.height);
}
```

Step 3 - Defining Accessors

Public accessors are added to the graphic object. These accessors deal with thickness and color. They appear in **bold** type in the following code sample.

```
/**
 * Changes the thickness of the shadow ellipse
 * @param thickness the new thickness
 */
public void setThickness(int thickness)
{
  this.thickness = thickness;
}

/**
 * Returns the thickness of the shadow ellipse
 * @return the thickness of the object.
 */
public int getThickness()
{
  return thickness;
}

/**
 * Changes the color of the ellipse.
 * @param color the new color.
 */
public void setColor(Color color)
{
  this.color = color;
}

/**
 * Returns the color of the shadow ellipse
 * @return the color of the object.
 */
public Color getColor()
{
  return color;
}

/**
 * Changes the color of the shadow
 * @param color the new color
 */
public void setShadowColor(Color color)
{
  this.shadowColor = color;
}

/**
 * Returns the color of the shadow
 * @return the color of the shadow
 */
public Color getShadowColor()
{
  return shadowColor;
}
```

Overriding IlvGraphic Methods

`IlvGraphic` is an abstract class. Therefore, some of its methods must be redefined in derived classes. This is the case for the following:

```
public abstract void draw(Graphics dst, IlvTransformer t)

public abstract IlvRect boundingBox(IlvTransformer t)

public abstract void applyTransform(IlvTransformer t)

public abstract IlvGraphic copy()
```

The other methods, such as `move`, `resize`, `rotate`, and `contains`, have a default implementation from the `IlvGraphic` class. These methods, as well as any other method that modifies the bounding box, are implemented by means of a call to the `applyTransform` function. If the new class has a parent that defines some of these methods, you can simply inherit the functions from this parent class.

The `ShadowEllipse` class defines the `draw`, `contains`, and `boundingBox` methods. In addition, it defines a `write` method that is necessary to write the object to a stream and a constructor that takes an `IlvInputStream` as a parameter to read the object from a stream. For details, see *The write Method* on page 42 and *The read Constructor* on page 42. These methods have no default implementation. You must provide a version of them for each subclass of the `IlvGraphic` class. If you do not intend to write additional information to the stream, you do not need to implement the `write` method, but you always need to define a constructor with an `IlvInputStream` parameter. Otherwise, you will not be able to read the object from a stream.

The draw Methods

This method demonstrates that drawing an object is merely a call to some of the primitive methods contained in the AWT `Graphics` class.

```
/**
   * Draws the object.
   * We override the draw method to define the way the object will appear.
   * @param dst The AWT object that will perform the
   * drawing operations.
   * @param t This parameter is the transformer used to draw the object.
   * This parameter may be a translation, a zoom or a rotation.
   * When the graphic object is drawn in a view (IlvManagerView),
   * this transformer is the transformer of the view.
   */
  public void draw(Graphics dst, IlvTransformer t)
  {
    // We first copy the rectangle that defines the bounding
    // rectangle of the object because we do not want to modify it.

    IlvRect r = new IlvRect(drawrect);

    // To compute the bounding rectangle of the object in
    // the view coordinate system, we apply the transformer ’t’
    // to the definition rectangle.
    // The transformer may define a zoom, a translation or a rotation.

    // We are using applyFloor so the resulting rectangle
    // is correctly projected for drawing in the view.
    // The object’s coordinate system is defined by ’float’ values
    // and we need ’int’ values to be able to draw. applyFloor will
    // apply the transformation to ’r’ and then calls Math.floor to
    // translate ’float’ values to ’int’ values.
    if (t != null)
      t.applyFloor(r);
    else
      r.floor();

    // The variable ’r’ now contains the bounding rectangle
    // of the object in the view’s coordinate system and we will
    // draw in this rectangle. In this rectangle, we will draw
    // two ellipses. We first draw the shadow ellipse on the
    // bottom right corner of the definition rectangle, then the main
    // ellipse on the top-left corner. Each ellipse will be of size
    // (r.width-thickness, r.height-thickness).

    int thick = thickness;

    // Computes a correct value for thickness.
    // Since we want the size of the ellipses
    // to be (r.width-thickness, r.height-thickness), we
    // must check that the thickness is not too big.
    if ((r.width <= thick) || (r.height <= thick))
      thick = (int)Math.min(r.width, r.height);

    // Sets the size of the ellipses.
    r.width  -= thick;
    r.height -= thick;

    // ’r’ now contains the bounding area of the main ellipse.

    // Computes a rectangle to draw the shadow.
    // We copy the variable ’r’, we will need it for the
    // second ellipse.
    IlvRect shadowRect = new IlvRect(r);
    shadowRect.translate(thick, thick);

    // Draws the shadow ellipse
    dst.setColor(getShadowColor());
    dst.fillArc((int)shadowRect.x,
                (int)shadowRect.y,
                (int)shadowRect.width,
                (int)shadowRect.height,
                0, 360);

    // Draws the main ellipse.
    dst.setColor(getColor());
    dst.fillArc((int)r.x,
                (int)r.y,
                (int)r.width,
                (int)r.height,
                0, 360);
  }
```

The method `draw` fills the two ellipses. The bounding rectangle, `drawrect`, actually covers both ellipses.

> ***Note:*** *The AWT methods, such as* `fillArc`*, require all coordinates to be integers. In ILOG JViews, however, the bounding box of a graphic object is defined by* `float` *values. To convert coordinates from* `float` *to* `int`*, we use the* `applyFloor` *and* `floor` *methods of the* `IlvTransformer` *class. You must use the same technique to ensure that the other objects comply with the library.*

The boundingBox Method

The method boundingBox simply transforms the bounding box. Note that it creates a copy of the rectangle drawrect even if the transformer is null. The reason is that the returned rectangle can be modified by ILOG JViews.

```
/**
  * Computes the bounding rectangle of the graphic
  * object when drawn with the specified transformer.
  */
 public IlvRect boundingBox(IlvTransformer t)
 {
   // We first copy the definition rectangle
   // because we do not want to modify it.
   IlvRect rect = new IlvRect(drawrect);
   // Apply the transformer on the rectangle to
   // translate to the correct coordinate system.
   if (t != null) t.apply(rect);
   return rect;
 }
```

The contains Method

The `contains` method returns `true` if the specified point is located within the main ellipse. All the coordinates are specified relative to the coordinate system of the view.

```
/**
  * Tests whether a point lies within the shape of the object.
  * This method will be called when you click on the object.
  * @param p The point where user clicks in the object’s coordinate system.
  * @param tp Same point as ’p’ but transformed by transformer ’t’
  * @param t The transformer used to draw the object.
  */
 public boolean contains(IlvPoint p, IlvPoint tp,
                         IlvTransformer t)
 {
   // We want to allow the user to click on the main ellipse
   // but not on the shadow ellipse.
   // This method will return true when the clicked point is
   // on the main ellipse.
   // We first compute the bounding rectangle of the main
   // ellipse in the view coordinate system, just like in the
   // method draw.
   IlvRect r = new IlvRect(drawrect);

   if (t != null)
     t.apply(r);

   int thick = thickness;

   if ((r.width <= thick) || (r.height <= thick))
     thick = (int)Math.min(r.width, r.height);

   r.width  -= thick;
   r.height -= thick;

   // Then we call PointInFilledArc that will return true
   // if the point is in the ellipse. ’r’ and ’tp’ are both
   // in the view coordinate system.
   return IlvArcUtil.PointInFilledArc(tp, r, (float)0, (float)360);
 }
```

Saving and Loading the Object Description

This section explains the input/output methods. To save and read an object in an ILOG JViews formatted file, you need to implement the `write` method and a constructor that takes an `IlvInputStream` parameter.

3. Graphic Objects

The write Method

The method `write` writes the colors of the object, the dimensions of the rectangle, and the thickness of the shadow to the provided output stream:

```
/**
  * Writes the object to an output stream.
  */
 public  void write(IlvOutputStream stream)
   throws IOException
 {
   // Calls the super class method that will write
   // the fields specific to the super class.
   super.write(stream);
   // Writes the colors.
   stream.write("color", getColor());
   stream.write("shadowColor", getShadowColor());
   // Writes the thickness.
   stream.write("thickness", getThickness());
   // Writes the definition rectangle.
   stream.write("rectangle", drawrect);
 }
```

In the `write` method, the `write` method of the superclass is called to save the information specific to the superclass. Then the `write` methods of the class `IlvOutputStream` are used to save the information specific to the class.

The read Constructor

To read a graphic object from a file, you must provide a specific constructor with an `IlvInputStream` parameter. This constructor must be public to allow the file reader to call it. Also, the constructor must read the same information, and in the same order, as that written by the `write` method.

```
/**
  * Reads the object from an IlvInputStream
  * @param stream the input stream.
  * @exception IlvReadFileException an error occurs when reading.
  */
 public ShadowEllipse(IlvInputStream stream) throws
                                                IlvReadFileException
 {
   // Calls the super class constructor that reads
   // the information for the super class in the file.
   super(stream);
   // Reads the color.
   setColor(stream.readColor("color"));
   // Reads the shadow color.
   setShadowColor(stream.readColor("shadowColor"));
   // Reads the thickness
   setThickness(stream.readInt("thickness"));
   // reads the definition rectangle.
   IlvRect rect = stream.readRect("rectangle");
   drawrect.reshape(rect.x, rect.y, rect.width, rect.height);
 }
```

The above constructor calls the read constructor of the superclass which reads the information specific to the superclass from the stream object. The subclass can then read its own information. The constructor uses the `read` methods defined in the class `IlvInputStream`.

Named Properties

Another kind of user property, called named property, can also be set on a graphic object. A named property is an instance of the class `IlvNamedProperty`. This class is an abstract class, it must be subclassed for your own needs. The difference between a named property and a user property is mainly that this type of property is named and that it can be saved with the graphic object in the `.ivl` file when the graphic object is saved. Note that a named property can also be stored in the manager or in a *layer* object, which is described in the section *Layers* on page 48.

To store a named property in a graphic object, use:

```
void setNamedProperty(IlvNamedProperty)
```

To get a named property, use:

```
void getNamedProperty(String name)
```

Here is an example of a named property:

```
import ilog.views.*;
import ilog.views.io.*;

public class MyNamedProperty extends IlvNamedProperty
{
  String value;

  public MyNamedProperty(IlvInputStream stream) throws IlvReadFileException
  {
    super(stream);
    this.value = stream.readString("value");
  }

  public MyNamedProperty(String name, String value)
  {
    super(name);
    this.value = value;
  }

  public MyNamedProperty(MyNamedProperty source)
  {
    super(source);
    this.value = source.value;
  }

  public IlvNamedProperty copy()
  {
    return new MyNamedProperty(this);
  }

  public boolean isPersistent()
  {
    return true;
  }

  public void write(IlvOutputStream stream) throws IOException
  {
    super.write(stream);
    stream.write("value", value);
  }

 }
```

This named property defines a member variable value of type `String` to store the value of the property.

Several methods have been created to allow the storage of the property in an `.ivl` file:

- The method `isPersistent` of the named property returns `true`.
- The method `write` is used to store the object in an `.ivl` file.

  This method is overridden to store the string in the member variable `value`.
  Note that the `super.write` call is mandatory for a correct storage of the property.

- The class defines a public constructor with an `IlvInputStream` parameter.

  This constructor is used to reload the property from the `.ivl` file.

Note: *The complete name of the class (including the name of the package) will be stored in the* `.ivl` *file. Therefore, if you change the name of the class, the property can no longer be loaded. Also, the class must be a public class to be saved in an* `.ivl` *file. Otherwise, it is impossible for the* `.ivl` *file reader to instantiate the class.*

CHAPTER

4

Managers

This section describes how to coordinate a large quantity of graphic objects through the use of a manager, that is, through the `IlvManager` class and its associated classes. You will find the following topics in this section:

- *Introducing Managers*
- *Binding Views to a Manager*
- *Managing Layers*
- *Managing Graphic Objects*
- *Selection in a Manager*
- *Managing Input Events*
- *Saving and Reading*

Introducing Managers

A manager is the data structure that contains the graphic objects. It organizes graphic objects in multiple storage places and coordinates the interactions between the display of graphic objects in multiple views, as illustrated in the following figure:

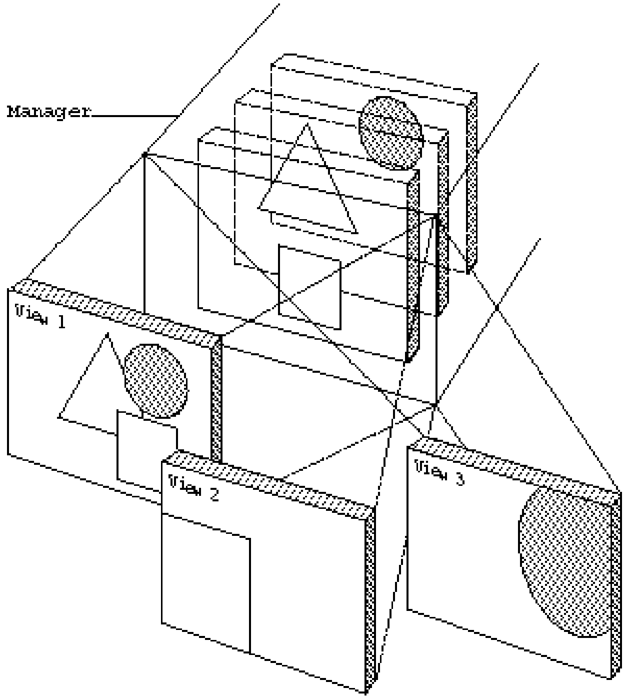


Figure 4.1 Manager Concept

Manager Views

A manager view is the AWT component where the graphic objects of a manager are displayed. To display graphic objects contained in a manager, you create at least one view, and often multiple views. The manager lets you connect as many views as you require to display graphic objects. The creation of a view is shown in *Creating a Manager and a View* on page 50.

A geometric transformation can be associated with each view so that you can display any portion of the global space where your graphic objects are located with appropriate scales (zooming) and rotations for each view. See *View Transformation* on page 52.

Layers

Instances of the `IlvManager` class handle a set of graphic objects derived from the ILOG JViews class called `IlvGraphic`. To organize graphic objects in a manager you place them in multiple storage areas called *layers*. The different graphic objects stored throughout the manager all share the same coordinate system. For this reason, a manager is a tool designed to handle objects placed on different priority levels. “Priority level” here means that objects stored in a higher screen layer are displayed in front of objects in lower layers. Each graphic object stored in a layer is unique to that layer and can be stored only in that layer.

Handling Input Events: Interactors and Accelerators

All input events are handled by means of *interactors* or *accelerators*.

Interactors

An `IlvManager` instance responds to user actions according to the state of the manager when a certain input event occurs, and also according to the position and shape of the object that receives the event. `IlvManager` actions can be either global (that is, applied to a whole view through instances of classes derived from `IlvManagerViewInteractor`) or local (applied to an object or a set of objects in a view through instances of classes derived from `IlvObjectInteractor`). The manager associates an interactor object, that is, an instance of the `IlvManagerViewInteractor` class, with each view. This interactor object processes events that are intended for that particular view. If the manager has not associated an interactor object with a view, then each event is handled by the interactor object associated with the graphic object that received the event. In this case, the interactor object belongs to the `IlvObjectInteractor` class, which manages the events for a particular object.

Accelerators

If no object is indicated when the event is received, or, if it has no associated `IlvObjectInteractor`, the manager tries to apply an accelerator, which is a direct call of a user-defined action, such as the pressing of a certain key sequence. In fact, in certain situations, the best solution is to establish a generic action for all the objects associated with a single event sequence so that, for example, pressing Ctrl+Z causes the view to zoom. To do this, ILOG JViews allows you to associate direct actions with events. These actions, which are bound neither to the view nor to the object that was clicked, are called *accelerators*.

Input/Output

The `IlvManager` class has a set of methods to read and write graphic object descriptions to a file. Manager properties, such as the layer or name of an object, can also be read and written.

Binding Views to a Manager

Attaching multiple views to a manager allows your program to display graphic objects simultaneously in various configurations.

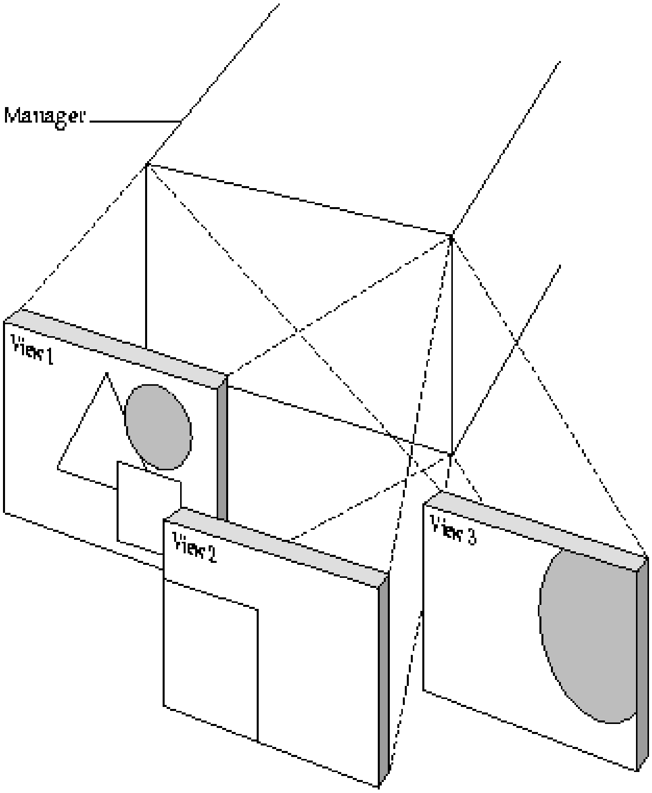


Figure 4.2 *Multiple Views Bound to a Manager*

To bind a view to a manager, you simply need to create a manager view, an instance of the class `IlvManagerView`. The constructors of `IlvManagerView` take an `IlvManager` parameter.

Creating a Manager and a View

The following code creates a manager and a view:

```
Frame frame= new Frame("JViews");
IlvManager mgr = new IlvManager();
IlvManagerView view = new IlvManagerView(mgr);
frame.add("Center", view);
frame.setSize(200,200);
frame.setVisible(true);
```

The class `IlvManagerView` is a subclass of the AWT class `java.awt.Container`.
A manager view is automatically disconnected from the manager when it is removed from its parent container.

To obtain a list of all the views attached to a manager, use the following `IlvManager` method:

```
Enumeration getViews()
```

You may also retrieve and change the manager displayed by a particular view using the following methods of the class `IlvManagerView`:

```
IlvManager getManager()
```

```
void setManager(IlvManager manager)
```

Listener on the Views of a Manager

This section presents the listeners to events constituting changes to a manager view.

ManagerViewsChangedEvent

When a view is attached or detached from a manager, a `ManagerViewsChangedEvent` event is fired by the manager. A class must implement the `ManagerViewsChangedListener` interface to be notified that an `IlvManagerView` has been attached or detached from the manager. This interface contains only the `viewChanged` method, which is called for each modification:

```
void viewChanged(ManagerViewsChangedEvent event)
```

To be notified, a class implementing this interface must register itself using the following method of the class `IlvManager`:

```
void addManagerViewsListener(ManagerViewsListener l)
```

ManagerChangedEvent

When the manager displayed by a view changes, as a result to a call to `setManager` on the view, the view fires a `ManagerChangedEvent`. A class must implement the `ManagerChangedListener` interface in order to be notified that the manager of the view has changed and must register itself on the view using the `addManagerChangedListener` method of the `IlvManagerView`. You can also specify that the listener no longer be notified of such events using the `removeManagerChangedListener` method.

When the manager of a view changes, the view calls the `managerChanged` method of the listeners.

```
void managerChanged(ManagerChangedEvent event)
```

This method is called with an instance of the class `ManagerChangedEvent` as a parameter containing information on the new and the old manager.

View Transformation

Each manager view (class `IlvManagerView`) has its own *transformer* to define the area of the manager that the view is displaying and also to define the zoom level and rotation applied to objects.

You may retrieve the current transformer of a view using the following method:

```
IlvTransformer getTransformer()
```

To modify the transformer associated with a view, use the following methods:

```
void setTransformer(IlvTransformer t)

void addTransformer(IlvTransformer t)

void translate(float deltax, float delaty, boolean redraw)

void zoom(IlvPoint, double, double, boolean)

void fitTransformerToContent()

void ensureVisible(IlvPoint p)

void ensureVisible(IlvRect rect)
```

To avoid distorting the image when it is zoomed in or out, you can specify that the vertical and horizontal aspect ratio remain the same by using the following methods:

```
boolean isKeepingAspectRatio()

void setKeepingAspectRatio(boolean set)
```

When the `KeepingAspectRatio` property is on, the view ensures that the horizontal and vertical scaling are always the same, whatever transformer you set in the view.

Example: Zooming a View

The following code zooms a view in by a scale factor of 2:

```
managerView.zoom(point, 2.0, 2.0, true);
```

The `point` given as an argument keeps its position after the zoom. The last parameter forces the redrawing of the view.

Transformer Listeners

When the transformer of a view changes, the view fires a `TransformerChangedEvent` event. A class must implement the `TransformerListener` interface to be notified that the transformer of the view has changed, and must register itself using the `addTransformerListener` method of `IlvManagerView`. You can also specify that the listener no longer be notified of such events using the `removeTransformerListener` method.

When the transformer of a view changes, the view calls the `transformerChanged` method of all listeners.

```
void transformerChanged(TransformerChangedEvent event)
```

This method is called with an instance of the class `TransformerChangedEvent` as a parameter. The `event` parameter can be used to retrieve the old and the new value of the transformer.

Scrolled Manager View

The library provides a convenience class that handles a manager view with two scroll bars in an AWT application: `IlvScrollManagerView`. This class automatically adjusts the scroll bars according to the area defined by the graphic objects contained in the manager. An equivalent object exists to be integrated into a Swing application: `IlvJScrollManagerView`.

Managing Double Buffering

Double buffering is a technique that is used to prevent the screen from flickering in an unpleasant manner when many objects are being manipulated. Since the manager view is implemented as a lightweight component, that is, as a direct subclass of `java.awt.Container`, it cannot handle double-buffering by itself. To use double-buffering in an AWT environment, the manager view must be the child of a heavyweight component, specially designed to handle double-buffering for instances of `IlvManagerView`. These components can be of the `IlvManagerViewPanel` or of the `IlvScrollManagerView` class.

The methods of the `IlvManagerViewPanel` and the `IlvScrollManagerView` class that handle double-buffering are:

```
boolean isDoubleBuffering()
```

```
void setDoubleBuffering(boolean set)
```

In a Swing application, you can directly use the `setDoubleBuffering` method of the class `IlvManagerView`. Putting an `IlvManagerView` into an `IlvJManagerViewPanel` or an `IlvJScrollManagerView` automatically enables the double-buffering on the manager view.

Example: Using Double-Buffering

This example creates a standard `IlvManagerView`, associates it with an `IlvManagerViewPanel`, and sets the double-buffering mode:

```
IlvManager mgr = new IlvManager();
IlvManagerView v = new IlvManagerView(mgr);
IlvManagerViewPanel panel = new IlvManagerViewPanel(v);
panel.setDoubleBuffering(true);
```

The Manager View Grid

Most editors provide a snapping grid that forces the objects to be located at specified locations. The coordinates where the end user can move the objects are called *grid points*. The class `IlvGrid` provides this functionality.

An instance of the class `IlvGrid` can be installed on each manager view. The view provides methods to set or retrieve the grid:

```
public void setGrid(IlvGrid grid)
```

```
public IlvGrid getGrid()
```

The following code installs a grid on a view with a vertical and horizontal grid point spacing of 10. The last two parameters are set to `true` to specify that the grid is visible and active:

```
mgrview.setGrid(new IlvGrid(Color.black, new IlvPoint(), 10f, 10f, true,
true));
```

When a grid is installed on a view, the standard ILOG JViews editing interactors, such as those for creating, moving, or editing an object, snap objects to the grid automatically.

These operations are not performed by the manager, but by the interactor itself. If you want to implement this mechanism in a new interactor you create, use the following method of the `IlvManagerView` class in the code of your new interactor:

```
public final void snapToGrid(IlvPoint point)
```

This method moves the `IlvPoint` argument to the closest point on the grid if a grid is installed and active. Otherwise, it does nothing.

Managing Layers

Layers are storage places for graphic objects in a manager. Each layer, as well as its graphic objects, are unique to a single *manager* and can only be controlled by this manager.

When you store graphic objects in layers, you indicate their placement throughout multiple layers. When you display graphic objects stored in multiple layers, you present layer contents in a series of one or several views, with each view controlled by and specific to the same manager.

Various methods let you manipulate these layers or the objects that they own. When redrawing takes place, a layer with the number N is placed in front of layers with numbers from N-1 to zero.

Inherent to the notion of layers is the concept of visual hierarchy among graphic objects stored in layers and displayed in views. In general, graphic objects of a more static nature, such as objects that might serve as shading or background for your ILOG JViews programs, should be put in a lower layer of the manager. Those graphic objects of a dynamic nature, such as objects with which users interact, should typically be put in a higher layer so that they are not hidden.

Setting Up Layers

Layers are handled internally by the class `IlvManagerLayer`. Layers can be accessed by their index or by their instance (a pointer to an `IlvManagerLayer`). By default, a manager is created with one layer. However, you can specify how many layers you want to create for a manager in the second parameter of the `IlvManager` constructor.

Once the manager has been created, you can modify the number of layers using the following methods:

```
void addLayer(int index)
```

```
void removeLayer(int index, boolean redraw)
```

and retrieve the number of layers with:

```
int getLayersCount()
```

Layers and Graphic Objects

When an object is added to a manager, you can specify the index of the layer where it should be inserted.

The following method adds the specified graphic object to the specified layer:

```
void addObject(IlvGraphic obj, int layer, boolean redraw)
```

To retrieve the index of the layer that contains a certain graphic object, use the following method:

```
int getLayer(IlvGraphic obj)
```

To change the layer, use the following method:

```
void setLayer(IlvGraphic obj, int newLayer, boolean redraw)
```

There are two essential properties that you can specify for the objects within a layer: visibility and selectability.

- **Visibility** With the following methods, you can indicate whether the objects within a certain layer should be visible to the user:

  ```
  void setVisible(int layer, boolean value, boolean redraw)
  ```

  ```
  boolean isVisible(int layer)
  ```

  You can also decide whether a layer is visible or not within a particular view. Refer to the following methods:

```
void setVisible(IlvManagerView view, int layer, boolean set,
boolean redraw)
```

```
boolean isVisible(IlvManagerView view, int layer)
```

Finally, you can have a visible layer in a view temporarily hide itself depending on certain conditions, generally depending on the zoom factor. This can be achieved through an `IlvLayerVisibilityFilter` that is called each time the `IlvManager` needs to redraw a layer. You should implement this interface and return whether or not the layer is visible with the `isVisible` method. To be active, this filter must be registered on the corresponding `IlvManagerLayer` using the `addVisibilityFilter` method.

- **Selectability** You can specify whether objects within a layer can be selected or not using the following methods. Objects that cannot be selected cannot be modified:

```
boolean isSelectable(int layer)
```

```
void setSelectable(int layer, boolean v)
```

For more methods dealing with layers, see `IlvManager` in the *ILOG JViews Reference Manual*.

Listener for Layer Changes in a Manager

A class must implement the `ManagerLayerListener` interface to be notified that layers have been inserted, removed or moved in a manager. This interface contains four methods:

- `void layerInserted(ManagerLayerInsertedEvent event)`

  which is called when a layer is added to a manager.

- `void layerMoved(ManagerLayerMovedEvent event)`

  which is called when a layer is moved in a manager.

- `void layerRemoved(ManagerLayerRemovedEvent event)`

  which is called when a layer is removed from a manager.

- `void layerChanged(ManagerLayerEvent event)`

  which is called for other changes in a layer.

To be notified of layer modifications, a class implementing this interface must register itself using the following method of the class `IlvManager`:

```
void addManagerLayerListener(ManagerLayerListener l)
```

Triple Buffering Layers

Certain applications can use the notion of layers to display a static background on top of which “live” graphic objects will be drawn and manipulated by the user.

In this type of application, the graphic objects taking part in the background are static and are not modified by the user or the application. Thus, it is possible to draw just once for all of the layers constituting the graphic background. This increases the drawing speed of the application.

The process is called *triple buffering*. This term is used because the layers will be drawn in an additional offscreen image. Thereafter, when the view needs to be redrawn, this image is used instead of redrawing the graphic objects.

> ***Note:*** *Unlike double buffering, triple buffering is not engaged to remove flickering but to increase the drawing speed. Double and triple buffering can be used together.*

For an instance of `IlvManagerView`, it is possible to indicate that a certain number of layers will be part of the triple buffering.

This is done using the following method of `IlvManagerView`:

```
void setTripleBufferedLayerCount(int n)
```

When this method is called, layers with indices between 0 and n-1 (the nth background layers) will be triple buffered.

Once the method is called and the view has been painted once, further modifications to graphic objects will not be rendered on the screen, since only the triple buffer image will be displayed.

Note that the triple buffer will be updated only when:

- The transformer of the view changes (you are zooming or panning the view).
- You add/remove layers.
- You change the number of triple buffered layers.

If for any reason you need to update the triple buffer, you can use the method:

```
void invalidateTripleBuffer(boolean repaint)
```

To summarize, an application will use triple buffering when the contents of background layers are static, and when the application does not require the user to zoom and pan frequently.

Managing Graphic Objects

The purpose of the manager is to manage a large set of graphic objects. Each graphic object can be managed by only one manager at a time, which means that you cannot add the same graphic object to two different managers.

Adding and Removing Objects

The following methods allow you to add a graphic object to a manager.

```
void addObject(IlvGraphic obj, int layer, boolean redraw)
```

```
void addObject(IlvGraphic obj, boolean redraw)
```

The following is an example that creates a rectangle object and adds it to a manager:

```
IlvManager mgr = new IlvManager();
IlvGraphic obj = new IlvRectangle(new IlvRect(10,10,100,100));
mgr.addObject(obj, false);
```

As for the layer the object is stored in, the second `addObject` method does not specify the layer where the object must be inserted. The reason for this is that there is a default insertion layer which allows you to add objects without specifying the layer at every call. The initial value for the default insertion layer is 0 but can be modified using the following methods:

```
int getInsertionLayer()
```

```
void setInsertionLayer(int layer)
```

Once an object has been added to a manager, you can remove it with:

```
void removeObject(IlvGraphic obj, boolean redraw)
```

You can also remove the objects from the manager, or from a specific layer using one of the following methods:

```
void deleteAll(boolean redraw)
```

```
void deleteAll(int layer, boolean redraw)
```

The following method can be used to know whether a graphic object is managed by the current manager:

```
boolean isManaged(IlvGraphic obj)
```

You can access all the objects of the manager or of a specified layer using one of the following methods:

```
IlvGraphicEnumeration getObjects()
```

```
IlvGraphicEnumeration getObjects(int layer)
```

These methods return an `IlvGraphicEnumeration` object facilitating the enumeration of the contents of the manager (or layer). You may use it in the following manner:

```
IlvGraphicEnumeration objects = manager.getObjects();
IlvGraphic obj;
while(objects.hasMoreElements()) {
    obj = objects.nextElement();
    //perform some action
}
```

> ***Note:*** *When stepping through the contents of the manager (or layer) by means of an enumeration, you must not modify the contents of the manager by adding or removing objects or layers. Doing so may lead to unpredictable results.*

Other useful methods will give you the number of objects in the manager or in a layer:

```
int getCardinal()
```

```
int getCardinal(int layer)
```

Modifying Geometric Properties of Objects

For every operation that leads to a modification of the bounding box of a graphic object, you must use the `applyToObject` method of the `IlvManager` class. As this method notifies the manager of the modification of the bounding box, you never directly call the `move` and `reshape` methods of `IlvGraphic`. For these basic operations, the manager has methods that call `applyToObject` for you:

```
void moveObject(IlvGraphic obj, float x, float y, boolean redraw)
```

```
void reshapeObject(IlvGraphic obj, IlvRect newrect, boolean
redraw)
```

Example: Moving an Object

The following code gets a reference to an object named `test` from the manager. If the object exists, it is moved to the point (10, 20) and redrawn (fourth parameter set to `true`).

```
IlvGraphic object = manager.getObject("test");
if (object != null)
  manager.moveObject(object, 10, 20, true);
```

The `moveObject` method is equivalent to the following code:

```
manager.applyToObject(object,
  new IlvApplyObject()
  {
   public void apply(IlvGraphic obj, Object arg){
     IlvPoint p = (IlvPoint) arg;
     obj.move(p.x, p.y);
   }
  },
  new IlvPoint(10,20), true);
```

Here we call the `applyToObject` method with `object` as a parameter and an anonymous class that implements the `IlvApplyObject` interface. The `arg` parameter is an `IlvPoint` that gives the new location of the object.

The method `applyToObject` is defined in the `IlvGraphicBag` interface, so you may call `applyToObject` directly from a graphic object using:

```
obj.getGraphicBag().applyToObject(obj, ...);
```

Applying Functions

To apply a user-defined function to objects that are located either partly or wholly within a specific region, use the following `IlvManager` methods:

- `void mapInside(IlvApplyObject f, Object arg, IlvRect rect, IlvTransformer t)`

  to apply a function to all graphic objects inside a specified rectangle.

- `void mapIntersects(IlvApplyObject f, Object arg, IlvRect rect, IlvTransformer t)`

  to apply a function to all graphic objects that intersect a specified rectangle.

Editing and Selecting Properties

The `IlvManager` class contains the following methods, which allow you to control the editing and selecting properties of an object added to a manager:

- To specify whether an object can be moved:

  `void setMovable(IlvGraphic obj, boolean v)`

  `boolean isMovable(IlvGraphic obj)`

- To specify whether an object can be edited:

  `void setEditable(IlvGraphic obj, boolean v)`

  `boolean isEditable(IlvGraphic obj)`

- To specify whether an object can be selected:

  `void setSelectable(IlvGraphic obj, boolean v)`

  `boolean isSelectable(IlvGraphic obj)`

These properties can be specified for graphic objects that are handled by an `IlvSelectInteractor` object. An `IlvSelectInteractor` object allows objects to be interactively selected in the manager, to be moved around, and to have their graphic properties edited.

Optimizing Drawing Tasks

A special manager feature minimizes the cost of drawing tasks to be done after geometric operations have been performed. This is useful in situations where you want to see the results of your work. This is implemented by using a region of invalidated parts of the display, called the *update region*. The update region stores the appropriate regions before any modifications are carried out on objects, as well as those regions that are relevant after these modifications have been carried out for each view.

To successfully apply an applicable function, you must mark the regions where the objects are located as invalid, apply the function, and then invalidate the regions where the objects involved are now located (applying the function may change the location of the objects). This mechanism is greatly simplified by a set of methods of the `IlvManager` class. Regions to be updated are only refreshed when the method `reDrawViews` is called. This means that refreshing the views of a manager is done by marking regions to be redrawn in a cycle of `initReDraws` and `reDrawViews`.

These cycles can be nested so that only the last call to the method `reDrawViews` actually updates the display. The `IlvManager` methods that help you optimize drawing tasks are:

- `initReDraws`

  Marks the beginning of the drawing optimization operation by emptying the region to update for each view being managed. Once this step is taken, direct or indirect calls to a draw instruction are deferred. For every `initReDraws`, there should be one call to `reDrawViews`, otherwise, a warning is issued. Calls to `initReDraws` can be embedded so that the actual refresh only takes place when the last call to `reDrawViews` is reached.

- `invalidateRegion`

  Defines a new region as invalid, that is, this region will be redrawn later. Each call to `invalidateRegion` adds the region to the update region in every view.

- `reDrawViews`

  Sends the drawing commands for the whole update region. All the objects involved in previous calls to `invalidateRegion` are then updated.

- `abortReDraws`

  Aborts the mechanism of deferred redraws (for example, if you need to refresh the whole screen). This function resets the update region to empty. If needed, you should start again with an `initReDraws` call.

- `isInvalidating`

  Returns `true` when the manager is in an `initReDraws`/`reDrawViews` state.

This mechanism is used in the `applyToObject` method.

In fact the call:

```
manager.applyToObject(obj, func, userArg, true);
```

is equivalent to:

```
manager.initReDraws();
manager.invalidateRegion(obj);
manager.applyToObject(obj, func, userArg, false);
manager.invalidateRegion(obj);
manager.reDrawViews();
```

The `invalidateRegion` method works with the bounding box of the object given as a parameter. When an operation applied to the object modifies its bounding box, `invalidateRegion` must be called twice: once before and once after the operation. For example, for a move operation, you must invalidate the initial region where the object was before being moved and invalidate the final region so that the object can be redrawn. In other situations, such as changing the background, only the call after the operation is necessary.

Listener on the Content of the Manager

When the content of the manager changes, the manager will fire a `ManagerContentChangedEvent` event. Any class can listen for the modification of the content of the manager by implementing the `ManagerContentChangedListener` interface.

This interface contains only the `contentsChanged` method.

```
void contentsChanged(ManagerContentChangedEvent evt)
```

This method is called when an object is added to or removed from the manager, or when the visibility, the bounding box or the layer of a graphic object changes. A class that implements this interface will register itself by calling the `addManagerContentChangedListener` method of the manager.

A `ManagerContentChangedEvent` can be of several types depending on the type of modification in the manager. For each type, there is a corresponding subclass of the class `ManagerContentChangedEvent`. The type of the event can be retrieved with the `getType` method of the class. The list of these subclasses is indicated below along with the type of change in the manager that is responsible for it:

- `ObjectInsertedEvent` (type `OBJECT_ADDED`) A graphic object has been inserted. You can retrieve the graphic object that was inserted with the `getGraphicObject` method.
- `ObjectRemovedEvent` (type `OBJECT_REMOVED`) A graphic object has been removed. You can retrieve the graphic object that was removed with the `getGraphicObject` method.

- ObjectBBoxChangedEvent (type OBJECT_BBOX_CHANGED) The bounding box of a graphic object has changed. You can retrieve the graphic object concerned using the getGraphicObject method and the old and new bounding box with the getOldBoundingBox and getNewBoundingBox methods.
- ObjectLayerChangedEvent (type OBJECT_LAYER_CHANGED) A graphic object has changed layers. You can retrieve the graphic object concerned using getGraphicObject and the old and new layer using the getOldLayer and getNewLayer methods.
- ObjectVisibilityChangedEvent (type OBJECT_VISIBILITY_CHANGED) The visibility of a graphic object has changed. You can retrieve the graphic object concerned using getGraphicObject. The method isObjectVisible will tell you the new state of the object.

A listener will cast the event depending on the type:

```
public void contentsChanged(ManagerContentChangedEvent event)
{
  if (event.getType() == ManagerContentChangedEvent.OBJECT_ADDED) {
    ObjectInsertedEvent e = (ObjectInsertedEvent)event;
    IlvGraphic object = e.getGraphicObject();
    ....
  }
}
```

As ManagerContentChangedEvent events can be sent very often (especially when adding numerous objects as in the case of reading a file), the manager provides a way to notify the listeners that it is currently doing a series of modifications. In this case, the event will contain a flag telling the listener that the manager is currently performing several modifications. This flag can be tested using the isAdjusting method of the ManagerContentChangedEvent class. The manager will notify the listeners of the end of a series by sending a final ManagerContentsChangedEvent of type ADJUSTMENT_END.

Thus, a listener can decide to react to all global modifications, but not to all individual modifications using the following code:

```
public class MyListener implements ManagerContentsChangedListener
{
  public void contentsChanged(ManagerContentChangedEvent event)
  {
     if (!event.isAdjusting()) {
        // do something
     }
  }
}
```

When making numerous modifications in a manager, you may want to be able to notify the listeners in the same way. To do so, you will use the `setContentsAdjusting` method of the manager in the following way:

```
manager.setContentsAdjusting(true);
try {
 //add a lot of objects
} finally {
  manager.setContentsAdjusting(false);
}
```

All operations done between the two calls to `setContentsAdjusting` will fire a `ManagerContentChangedEvent` event with the `isAdjusting` flag set to `true`.

The call to the `setContentsAdjusting` method with the parameter set to `false` may send the file `ADJUSTMENT_END` event.

Using this mechanism can also help the internal listeners of ILOG JViews to work in a more efficient way, so we recommend that you use this mechanism.

Selection in a Manager

The manager allows you to select objects. To display selected objects within a manager, ILOG JViews creates *selection objects* which are drawn on top of the selected objects. To take an example, the selection object can be drawn as a set of handles around the selected object. Selection objects are stored in the manager. Unlike regular graphic objects, they are internally managed and cannot be manipulated.

Managing Selected Objects

When a graphic object is selected, a selection object is created and is drawn on top of the graphic object. Selection objects are subclass instances of the class `IlvSelection`. As such, they are also graphic objects. The class `IlvSelection` is an abstract class that has been subclassed to create several classes of selection objects specialized in the selection of specific graphic objects. For example, the class `IlvSplineSelection` is a selection object for the selection of an `IlvSpline` object.

The default selection object for graphic objects is an instance of the class `IlvDrawSelection`. This class draws eight handles around the object, one on each of the four sides and one on each corner.

To select or deselect a graphic object in a manager, use the `setSelected` method:

```
void setSelected(IlvGraphic obj, boolean select, boolean redraw)
```

Once an object has been selected, you can retrieve its selection object using:

```
IlvSelection getSelection(IlvGraphic obj)
```

This method returns `null` if the object does not have an associated selection object; in other words, if the graphic object is not selected. You can also use the following method to determine whether the object is selected or not:

```
boolean isSelected(IlvGraphic obj)
```

To obtain the selected object from the selection object, use the `getObject` method of `IlvSelection`.

You may obtain an enumeration of all the selected objects in the manager with:

```
IlvGraphicEnumeration getSelectedObjects()
```

You may use this method as follows:

```
IlvGraphicEnumeration selectedobjs = manager.getSelectedObjects();
IlvGraphic obj;

while(selectedobjs.hasMoreElements()) {
    obj = selectedobjs.nextElement();
    //perform some action
}
```

Note: *To avoid unpredictable results, you must not select or deselect graphic objects when stepping through the enumeration as in the example above.*

Other methods of `IlvManager` allow you to select and deselect all objects in the manager or in a particular layer:

```
void selectAll(IlvManagerView view, boolean redraw)
```

```
void selectAll(boolean redraw)
```

```
void deSelectAll(boolean redraw)
```

```
void deSelectAll(int layer, boolean redraw)
```

Selection Interactor

The library provides the `IlvSelectInteractor` class which allows you to select and deselect objects in an interactive way (using the mouse). It also allows you to edit graphic objects. For more information, see *The Selection Interactor* on page 78.

Creating Your Own Selection Object

The selection object depends on the graphic object. In fact, the manager creates the selection object using the following method of the graphic object:

```
IlvSelection makeSelection()
```

You may override this method to return your own instance of the selection object. Another possibility is to set an `IlvSelectionFactory` on the manager and let this factory decide which subclass of `IlvSelection` should be instantiated depending on the graphic object. The following is an example which creates a new selection object (a white border) around the selected object:

```
class mySelection extends IlvSelection
{
  static final int thickness = 3;
  mySelection(IlvGraphic obj)
  {
    super(obj);
  }

  public void draw(Graphics g, IlvTransformer t)
  {
    g.setColor(Color.white);
    IlvRect rect = boundingBox(t);
    for (int i = 0; i < thickness; i++) {
      if ((int)Math.floor(rect.width) >
           2*i && (int)Math.floor(rect.height) > 2*i)
         g.drawRect((int)Math.floor(rect.x)+i,
                    (int)Math.floor(rect.y)+i,
                    (int)Math.floor(rect.width)-2*i-1,
                    (int)Math.floor(rect.height)-2*i-1);
    }
  }

  public IlvRect boundingBox(IlvTransformer t)
  {
    // get the bounding rectangle of the selected object
    IlvRect bbox = getObject().boundingBox(t);
    bbox.x-= thickness;
    bbox.y-= thickness;
    bbox.width+= 2*thickness;
    bbox.height+= 2*thickness;
    return bbox;
  }

  public boolean contains(IlvPoint p, IlvPoint tp, IlvTransformer t)
  {
    return false;
  }
}
```

You can see that the selection object is defined in the same way as a graphic object. The constructor of a selection object always takes the selected object as a parameter. Note that the `boundingBox` method of the selection object uses the `boundingBox` method of the selected object so that the selection object (in this case, the white border) is always around the selected object, whatever the transformer is.

Listener on the Selections in a Manager

A class must implement the `ManagerSelectionListener` interface to be notified that selections in a manager have been modified. This interface contains only the `selectionChanged` method, which is called each time an object is selected or deselected.

```
void selectionChanged(ManagerSelectionChangedEvent event)
```

To be notified of selections and deselections, a class must register itself using the following method of the class `IlvManager`:

```
void addManagerSelectionListener(ManagerSelectionListener l)
```

Note that the `selectionChanged` method is called just after the object is selected or deselected, so you can easily determine whether it is a selection or a deselection. You do this the following way:

```
class MyListener implements ManagerSelectionListener
{
  public void selectionChanged(ManagerSelectionEvent event)
  {
    // retrieve the graphic object
    IlvGraphic obj  = event.getGraphic();
    IlvManager manager = event.getManager();
    if (manager.isSelected(obj)) {
      // object was selected
    } else {
      // object was deselected
    }
  }
}
```

When numerous objects are being selected, for example, as a result of the call to the `selectAll` method of the manager, many selection events will be sent to the selection listeners. This can be inefficient for some listeners that need to perform an action when the selection is stable. For example, a property inspector showing the properties of selected objects does not need to be updated for each individual selection when a number of objects are selected at the same time. To solve this kind of problem, the `ManagerSelectionChangedEvent` class has the:

- `isAdjusting` method that tells you if the event is part of a series of selection events.
- `isAdjustmentEnd` method which indicates that the event is the last one of a series.

In the case of our “property inspector,” the listener would be as follows:

```
class MyListener implements ManagerSelectionListener
{
  public void selectionChanged(ManagerSelectionChangedEvent event)
  {
    if (!event.isAdjusting() || event.isAdjustmentEnd())
    {
       // update the properties only if this is a single
       // selection or the end of a series.
    }
  }
}
```

You may want to use the same “adjustment” notification when selecting numerous objects in a manager. The `IlvManager` allows you to do this using the `setSelectionAdjusting` method:

```
boolean isAdjusting = manager.isSelectionAdjusting();
manager.setSelectionAdjusting(true);
try {
  // select or deselect a lot of objects.
} finally {
  manager.setSelectionAdjusting(isAdjusting);
}
```

Managing Input Events

There are two ways of handling the input events in a manager view:

- An interactor can be set to the view using the class `IlvManagerViewInteractor`, which will handle all the events that occur on a view.
- An *object interactor* on a graphic object can be used. This interactor, an instance of the class `IlvObjectInteractor`, handles the events occurring on a particular object if no view interactor has been installed on the view.

Object Interactors

When you want to associate a specific behavior to an object, you may use an object interactor (class `IlvObjectInteractor` and its subclasses). Whenever an event is received by a manager view that has no associated view interactor, the manager attempts to send it to an object by a call to an attached object interactor. If there is an object at the event location and, if this object is connected to an object interactor, the manager sends the event to that interactor. If the interactor does not manage this event, or, if the situation is not applicable, the manager tries to handle the event by means of accelerators.

You can create an `IlvObjectInteractor` instance and bind it to an object or a set of objects using the `IlvGraphic` method `setObjectInteractor`. As soon as this binding occurs, the object receives user events and deals with them, therefore, it is the interactor, and not the object itself, that manages these events.

Querying, setting, or removing an object interactor can be done by means of calls to the following methods on the `IlvGraphic` instance:

```
IlvObjectInteractor getObjectInteractor()

void setObjectInteractor(IlvObjectInteractor interactor)
```

An instance of `IlvObjectInteractor` can be shared by several graphic objects. This allows you to reduce the amount of memory needed to handle the same interaction on a large number of graphic objects. To share the same object interactor instance, do not use `new` to create your interactor. Use the `Get` method of the class `IlvObjectInteractor`.

Example: Extending the IlvObjectInteractor Class

The following example shows how to extend the `IlvObjectInteractor` class. The `MoveObjectInteractor` class defined in this example allows you to move an object using the mouse, moving it to where you release the mouse button.

The `MoveObjectInteractor` extends the `IlvObjectInteractor` class and defines four attributes.

The attribute `mrect` specifies the future bounding rectangle of the graphic object. This data is updated each time the object is dragged. The `dx` and `dy` attributes represent the translation from the original clicked point and the current top-left corner of the graphic object. The Boolean value `dragging` is set to `true` when the user starts dragging the object.

Events are processed by the `processEvent` method below:

```
protected boolean processEvent(IlvGraphic obj, AWTEvent event,
                               IlvObjectInteractorContext context)
{
   switch (event.getID())
   {
     case MouseEvent.MOUSE_PRESSED:
       return processButtonDown(obj, (MouseEvent)event, context);
     case MouseEvent.MOUSE_DRAGGED:
       return processButtonDragged(obj, (MouseEvent)event, context);
     case MouseEvent.MOUSE_RELEASED:
       return processButtonUp(obj, (MouseEvent)event, context);
     default:
       return false;
   }
}
```

The `processEvent` method dispatches events to three different methods depending on their type. The `processEvent` method takes a graphic object, an event, and a context as its parameters. The context `IlvObjectInteractorContext` is an interface that must be implemented by classes allowing the use of an object interactor. The class `IlvManager` takes care of that and passes a context object to the object interactor. From a context, you can get a transformer, change the mouse cursor, and so on.

In the processButtonDown method below, the distance between the clicked point and the current top-left corner of the graphic object is stored in the attributes dx and dy. Note that the positions are stored in the view coordinate system. The top-left corner of the graphic object is extracted from the bounding rectangle of the object, which is computed using the boundingBox method. The invalidateGhost method is then called. This method requests the interactor context to redraw the region corresponding to the current bounding rectangle (stored in mrect)..

```
public boolean
processButtonDown(IlvGraphic obj,
                  MouseEvent event,
                  IlvObjectInteractorContext context)
{
  if ((event.getModifiers() & InputEvent.BUTTON2_MASK) != 0 ||
      (event.getModifiers() & InputEvent.BUTTON3_MASK) != 0)
    return true ;
  if (dragging)
    return true ;
  dragging = true;
  IlvPoint p = new IlvPoint(event.getX(), event.getY());
  mrect = obj.boundingBox(context.getTransformer());
  dx = p.x - mrect.x;
  dy = p.y - mrect.y;
  mpoint.move(p.x -dx, p.y-dy);
  invalidateGhost(obj, context);
  return true;
}
```

The invalidateGhost method is implemented as follows:.

```
private void
invalidateGhost(IlvGraphic obj,
                IlvObjectInteractorContext context)
{
  if (obj == null || context == null)
    return;
  if (mrect == null || mrect.width == 0 || mrect.height == 0)
    return;
  IlvRect invalidRegion = new IlvRect(mrect);
  context.repaint(invalidRegion);
}
```

The principle for drawing and erasing the ghost is the following: the interactor invalidates regions of the context (the bounds of the ghost before and after any position change), while the drawing of the ghost is actually performed only when the interactor is requested to do so by the drawing system. Indeed, the method handleExpose, defined on the base class

IlvObjectInteractor, is called at each redraw of the view. The method, whose implementation on the base class does nothing, is overridden as follows: .

```
public void handleExpose(IlvGraphic obj,
                         Graphics g,
                                   IlvObjectInteractorContext context)
{
  drawGhost(obj, g, context);
}
```

The actual drawing of the ghost is done by the following drawGhost method :.

```
protected void drawGhost(IlvGraphic obj,
                         Graphics g,
                         IlvObjectInteractorContext context)
{
  if (mrect != null) {
    Graphics g = context.getGraphics();
    g.setColor(context.getDefaultGhostColor());
    g.setXORMode(context.getDefaultXORColor());
    IlvTransformer t = context.getTransformer();
    IlvRect r = obj.boundingBox(t);
    IlvTransformer t1 = new IlvTransformer(new IlvPoint(mrect.x - r.x,
                                                        mrect.y -r.y));
    t.compose(t1);
    obj.draw(g, t);
  }
}
```

The Graphics object that is passed as an argument is set to XOR mode using the default XOR color and ghost color defined in the context. Next, a transformer is computed that will draw the object in the desired location given by mrect. Note that the object is not translated, only drawn at another location. The method does nothing if mrect is null. This prevents drawing the ghost if the drawGhost method happens to be called after the end of the interaction.

Mouse dragged events are handled as follows:

```
protected boolean
processButtonDragged(IlvGraphic obj, MouseEvent event,
                     IlvObjectInteractorContext context)
{
  if (!dragging || mrect == null)
    return false;
  IlvPoint p = new IlvPoint(event.getX(), event.getY());
  invalidateGhost(obj, context);
  mrect.move(p.x - dx, p.y - dy);
  IlvTransformer t = context.getTransformer();
  if (t != null)
    t.inverse(mrect);
  context.ensureVisible(p);
  t = context.getTransformer();
  if (t != null)
    t.apply(mrect);
  invalidateGhost(obj, context);
  return true;
}
```

We first invalidate the current ghost by calling `invalidateGhost`. The new required location is changed to the position of the mouse, translated with the original translation:

```
mrect.move(p.x - dx, p.y - dy);
```

Then `ensureVisible` is called on the context. If the current dragged point is outside the visible area of the view, this will scroll the view of the manager so that the dragged point becomes visible. This operation may change the transformer of the view, as the value of `mrect` is stored in the coordinate system of the view. Before calling `ensureVisible`, we transform the value of `mrect` in the manager coordinate system by calling:

```
IlvTransformer t = context.getTransformer();
if (t != null) t.inverse(mrect);
```

After the call to `ensureVisible`, we transform the value of `mrect` back to the view coordinate system by calling:

```
t = context.getTransformer();
if (t != null) t.apply(mrect);
```

The actual moving of the graphic object is done when the mouse button is released. The mouse released event is handled like this:

```
protected boolean
processButtonUp(IlvGraphic obj,
                MouseEvent event,
                IlvObjectInteractorContext context)
{
  if (!dragging || mrect == null)
    return true;
  dragging = false;
  invalidateGhost(obj, context);
  doMove(obj, context);
  mrect = null;
  return true;
}
```

First the ghost is invalidated by calling the `invalidateGhost` method. Then the `doMove` method is called. This method updates the position of the graphic object according to the final coordinates of `mrect`. After moving the object, `mrect` is set to `null` to prevent further drawings of the ghost.

The implementation of the method `doMove` is the following::

```
void doMove(IlvGraphic graphic,
            IlvObjectInteractorContext context)
{
  if (mrect == null)
    return;
  IlvTransformer t = context.getTransformer();
  if (t != null)
    t.inverse(mrect);
  graphic.getGraphicBag().moveObject(graphic, mrect.x,
                                     mrect.y, true);
}
```

The value of `mrect` is translated in the coordinate system of the manager by calling:

```
IlvTransformer t = context.getTransformer();
if (t != null)
  t.inverse(mrect);
```

You should never try to change directly the position or the shape of a managed graphic object (or, more precisely, to modify its bounding box), for instance by calling directly methods of the graphic object. Such changes must be done through a function, in this case `moveObject`, which is applicable to managers and takes all the necessary precautions. For further information, see *Modifying Geometric Properties of Objects* on page 59.

View Interactors

The `IlvManagerViewInteractor` class handles view behavior. The role of this class is to handle complex sequences of user input events that are to be processed by a particular view object. You can add or remove a view interactor with the following methods:

```
IlvManagerViewInteractor getInteractor()

void setInteractor(IlvManagerViewInteractor inter)

void pushInteractor(IlvManagerViewInteractor inter)

IlvManagerViewInteractor popInteractor()
```

Predefined View Interactors

ILOG JViews provides predefined view interactors. Following is a list of these interactors:

- `IlvDragRectangleInteractor` - Draws a rectangle that can be used for several purposes. See the *Example: Implementing the DragRectangleInteractor Class* on page 75.
- `IlvMakeRectangleInteractor` - Allows the creation of `IlvRectangle` objects.
- `IlvMakeArcInteractor` - Allows the creation of `IlvArc` objects.
- `IlvMakeEllipseInteractor` - Allows the creation of `IlvEllipse` objects.
- `IlvMakeReliefRectangleInteractor` - Allows the creation of objects of the `IlvReliefRectangle` class.
- `IlvMakeRoundRectangleInteractor` - Allows the creation of objects of the `IlvRectangle` class with round corners.
- `IlvUnZoomViewInteractor` - Allows the unzooming command. You have to draw a rectangular region into which the area you are watching is unzoomed.
- `IlvZoomViewInteractor` - Allows the zooming command. You draw a rectangular region where you want to zoom.
- `IlvMakePolyPointsInteractor` - Allows the creation of polypoints objects.
- `IlvMakeLineInteractor` - Allows the creation of objects of the `IlvLine` class.
- `IlvMakeArrowLineInteractor` - Allows the creation of objects of the `IlvArrowLine` class.
- `IlvMakeLinkInteractor` - Allows the creation of objects of the `IlvLinkImage` class.
- `IlvMakePolyLinkInteractor` - Allows the creation of objects of the `IlvPolylineLinkImage` class.

- IlvMakePolygonInteractor - Allows the creation of objects of the IlvPolygon class.
- IlvMakePolylineInteractor - Allows the creation of objects of the IlvPolyline class.
- IlvMakeArrowPolylineInteractor - Allows the creation of objects of the IlvArrowPolyline class.
- IlvMakeSplineInteractor - Allows the creation of objects of the IlvSpline class.
- IlvEditLabelInteractor - Allows the creation and editing of objects that implement the IlvLabelInterface such as IlvLabel or IlvZoomableLabel.
- IlvMoveRectangleInteractor - Drags a rectangle and performs an action when releasing the mouse button.
- IlvSelectInteractor - Allows the selection and editing of graphic objects.
- IlvRotateInteractor - Allows the rotation of a selected graphic object.
- IlvPanInteractor - Allows the translation of a view without using scroll bars.
- IlvMagnifyInteractor - Allows the magnfication of part of the view under the mouse pointer.

Interactor Listeners

When the active interactor of a view changes, the view fires an InteractorChangedEvent event. A class must implement the InteractorListener interface in order to be notified that a view interactor has been modified and must register itself using the addInteractorListener method of IlvManagerView. You can also specify that the listener no longer be notified of such events by using the removeInteractorListener method.

When the interactor of a view changes, the view calls the interactorChanged method of the listeners.

```
void interactorChanged(InteractorChangedEvent event)
```

This method is called with an instance of the class InteractorChangedEvent as a parameter containing information on the new and the old interactor.

Example: Implementing the DragRectangleInteractor Class

We are now going to explain how the methods of the predefined view interactor IlvDragRectangleInteractor are implemented. You can use the following example as a starting point for creating your own interactor functionality. The IlvDragRectangleInteractor is used to specify a rectangular region in a view. When this rectangle is selected, the fireRectangleDraggedEvent method is called. This rectangle can then be used for various purposes in derived interactors. For example, we might create a subclass of this interactor that will allow the user to zoom in the selected area.

The `DragRectangleInteractor` class defines three attributes: `start`, `rectangle`, and `dragging`:

```
public class IlvDragRectangleInteractor extends
                                            IlvManagerViewInteractor {
  /**  The anchor point of the rectangle. */
  private final IlvPoint start = new IlvPoint();
  /**  The rectangle when dragging. */
  private final IlvRect rectangle = new IlvRect();
  /**  True if we are dragging. */
  private boolean dragging = false;
...
}
```

The attribute `start` is the point where we begin to drag, `rectangle` is the rectangle that is drawn when dragging, and `dragging` is a Boolean whose value is `true` when the user drags.

The `enableEvents` method called in the constructor takes the `MOUSE_EVENT_MASK` and `MOUSE_MOTION_EVENT_MASK` as parameters. Events must be enabled to be taken into account by the interactor:

```
public DragRectangleInteractor()
{
  enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                                 AWTEvent.MOUSE_MOTION_EVENT_MASK);
}
```

The `processMouseEvent` method handles the `MOUSE_PRESSED` and `MOUSE_RELEASED` events:

```
protected void processMouseEvent(MouseEvent event)
{
  switch (event.getID()) {
    case MouseEvent.MOUSE_PRESSED:
    {
      if (dragging) break;
      if ((event.getModifiers() & InputEvent.BUTTON2_MASK) == 0 &&
          (event.getModifiers() & InputEvent.BUTTON3_MASK) == 0)
      {
        dragging = true;
        IlvTransformer t = getTransformer();
        start.move(event.getX(), event.getY());
        t.inverse(_start);
        rectangle.width = 0;
        rectangle.height = 0;
      }
      break;
    }
case MouseEvent.MOUSE_RELEASED:
      if (dragging) {
        dragging = false;
        drawGhost();
        rectangle.width = 0;
        rectangle.height = 0;
          fireRectangleDraggedEvent(new IlvRect(rectangle), event);
      }
   }
}
```

When the mouse button is pressed, the mouse pointer coordinates are stored in the `start` variable and are converted for storage in the coordinate system of the manager. When the mouse is released, the `drawGhost` method of `IlvManagerViewInteractor` is called to erase the ghost image. The width and height of the rectangle are set to `0` to prevent further drawings of the ghost, and the `fireRectangleDraggedEvent` method is called to notify the end of the drag operation. The following code demonstrates the dragged rectangle.

> ***Note:*** *The* `drawGhost` *method can be used to perform a temporary drawing that gives the user feedback on the action of his present operation.*

The `processMouseMotionEvents` handles the `MOUSE_DRAGGED` events:

```
protected void processMouseMotionEvent(MouseEvent event)
{
  if (event.getID() == MouseEvent.MOUSE_DRAGGED && dragging) {
    drawGhost();
    IlvTransformer t = getTransformer();
    IlvPoint p = new IlvPoint(event.getX(), event.getY());
    ensureVisible(p);
    rectangle.reshape(start.x, start.y, 0,0);
    t.inverse(p);
    rectangle.add(p.x, p.y);
    drawGhost();
  }
}
```

We first erase the rectangle by calling `drawGhost`. The call to `ensureVisible` ensures that the dragged point remains visible on the screen. The new rectangle is then computed in the coordinate system of the manager and `drawGhost` is called to redraw the new rectangle.

The `drawGhost` method simply draws the dragged rectangle. Since the rectangle is in the manager coordinate system, the method needs to apply the view transformer before drawing:

```
protected void drawGhost(Graphics g)
{
  IlvRect rect = new IlvRect(rectangle);
  IlvTransformer t = getTransformer();
  if (t != null)
    t.apply(rect);
  if (rect.width > 0 && rect.height >0) {
    g.drawRect((int)Math.floor(rect.x), (int)Math.floor(rect.y),
               (int)Math.floor(rect.width),
               (int)Math.floor(rect.height));
  }
}
```

The Selection Interactor

The ILOG JViews library provides a predefined view interactor, `IlvSelectInteractor`, for selecting and editing graphic objects in a manager. This class allows you to:

- Select an object by clicking on it.
- Select or deselect several objects using Shift-Click.
- Select several objects by dragging a rectangle around them.
- Move one or several objects by selecting them and dragging the mouse.
- Edit objects by manipulating their selection object.

The interactor can be customized to your needs, as follows:

- You can enable or disable multiselection using:

  ```
  public void setMultipleSelectionMode(boolean v)
  ```

  ```
  public boolean isMultipleSelectionMode()
  ```

- You can select the mode for ‘selecting objects by dragging a rectangle around’: opaque or ghost:

  ```
  public void setOpaqueDragSelection(boolean o)
  ```

  ```
  public boolean isOpaqueDragSelection()
  ```

- You can select the mode for moving graphic objects: opaque or ghost:

  ```
  public void setOpaqueMove(boolean o)
  ```

  ```
  public boolean isOpaqueMove()
  ```

- You can select the mode for resizing graphic objects: opaque or ghost:

  ```
  public void setOpaqueResize(boolean o)
  ```

  ```
  public boolean isOpaqueDragSelection()
  ```

- You can select the mode for editing polypoints objects: opaque or ghost:

  ```
  public void setOpaquePolyPointsEdition(boolean o)
  ```

  ```
  public boolean isOpaqueDragSelection()
  ```

- You can specify the modifier that allows multiple selection:

  ```
  public void setMultipleSelectionModifier(int m)
  ```

  ```
  public boolean getMultipleSelectionModifier()
  ```

- You can specify the modifier that allows selection by dragging a rectangle starting from a point on top of a graphic object:

```
public void setSelectionModifier(int m)
```

```
public boolean getSelectionModifier()
```

- You can allow the selection of several objects using a dragged rectangle:

  ```
  public void setDragAllowed(boolean v)
  ```

  ```
  public boolean isDragAllowed()
  ```

- You can change the ability to move objects by:

  ```
  public void setMoveAllowed(boolean v)
  ```

  ```
  public boolean isMoveAllowed()
  ```

- You can change the ability to edit objects by:

  ```
  public void setEditionAllowed(boolean v)
  ```

  ```
  public boolean isEditionAllowed()
  ```

> ***Note:*** *The ability to move, edit, or select an object can be controlled object by object using the properties of the object.*

Many other customizations can be done by subclassing the interactor and overriding the appropriate method.

The editing of a graphic object is controlled by an object interactor. When a graphic object is selected, the manager may dispatch events occurring on the selection object to the *object interactor* attached to the selection object. This object interactor is created by the selection object with a call to the `getDefaultInteractor` method of the class `IlvSelection`. When creating your own selection object, you can also create an object interactor to edit the selected object. The default object interactor is the class `IlvReshapeSelection` that allows the reshaping of graphic objects by pulling on the handles of th.e objects.

Saving and Reading

The manager provides facilities to save its contents to a file. The resulting file is an ASCII or binary file in the `.ivl` format that contains information about the layers and the graphic objects. The saving methods are as follows:

```
void write(OutputStream stream, boolean binary) throws IOException
```

```
void write(String filename) throws IOException
```

```
void write(String filename, boolean binary) throws IOException
```

You can save data in either an ASCII or binary file, the binary format being, however, more compact and faster to read than the ASCII format.

```
void read(InputStream stream) throws IOException,
      IlvReadFileException
```

```
void read(String filename) throws IOException,
IlvReadFileException
```

```
void read(URL url) throws IOException, IlvReadFileException
```

The `read` methods may throw an exception in the following situations:

- The file is not an `.ivl` file.
- The `.ivl` format is not correct.
- A graphic class cannot be found.

Whether the `.ivl` file is an ASCII or binary file is detected automatically by the `read` methods.

You can save/read the information on your own graphic objects by providing the appropriate methods when creating your own graphic object class. For more information, see *Input/ Output* on page 49.

CHAPTER

5

Graphers

This section introduces the grapher, a high-level ILOG JViews functionality. With the grapher, you can create graphic programs that both include and represent hierarchical information with graphic objects. Based on the manager, the grapher is a natural extension of the manager concepts. The following topics are covered in this section:

- *Introducing Graphers*
- *Managing Nodes and Links*
- *Contact Points*
- *Grapher Interactor Class*
- *Creating a New Class of Link*

Introducing Graphers

A grapher is an instance of the `IlvGrapher` class, a subclass of the `IlvManager` class. The grapher library offers enhanced performance to create programs, including a large quantity of dynamic interconnected information, such as network management and file management programs. You have an extensive set of objects in the grapher library for creating:

- **Nodes**, the visual reference points in a hierarchy of information. A node is a graphic object, an instance of a subclass of the `IlvGraphic` class.

- **Links**, the visual representation of connections between nodes. Links are also graphic objects, instances of the `IlvLinkImage` class or its subclasses.

Here is an example of a grapher that connects graphic objects of the class `IlvReliefLabel`.

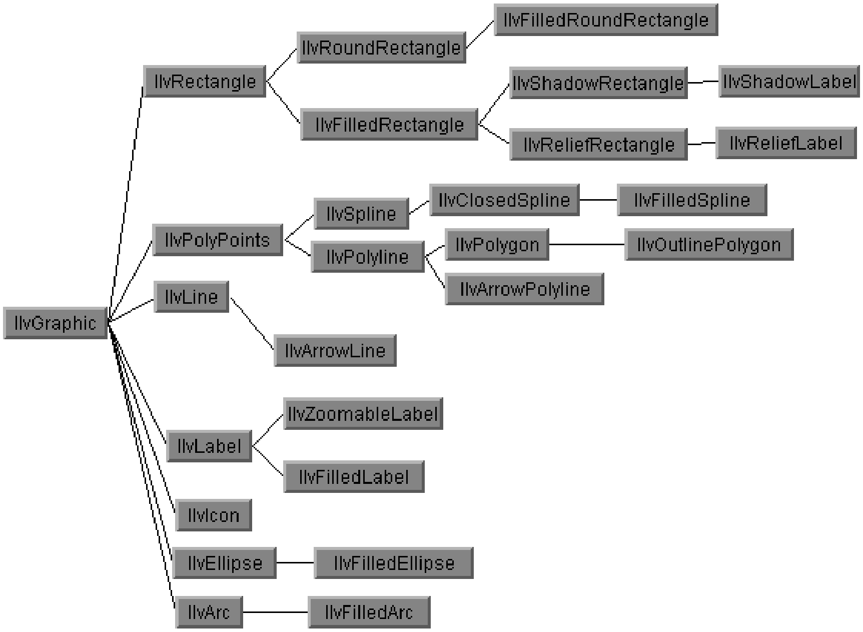


Figure 5.1 *Grapher Connecting Graphic Objects of IlvReliefLabel*

Managing Nodes and Links

A grapher is composed of nodes and links. Nodes are simple graphic objects, which means that any graphic object can be used as a grapher node. To add a node to a grapher, use one of the following methods:

```
void addNode(IlvGraphic obj, boolean redraw)
```

```
void addNode(IlvGraphic obj, int layer, boolean redraw)
```

These methods will add the object in the specified layer of a grapher and add additional information to the object so that it becomes a node. Graphic objects that have been added into the grapher using the `addObject` method can also become nodes of the grapher using the method:

```
void makeNode(IlvGraphic obj)
```

Basic Grapher Link Class

Nodes in a grapher are connected with links. All links are instances of the class `IlvLinkImage` (or subclasses). The constructor of the class `IlvLinkImage` has two graphic objects as parameters, so, when creating a link, you always have to give the origin and destination of the link. Here is the constructor of `IlvLinkImage`:

```
IlvLinkImage(IlvGraphic from, IlvGraphic to, boolean oriented)
```

The `oriented` parameter specifies whether or not an arrowhead is to be drawn at one end of the link. Once a link is created, you can add it to the grapher using one of the following methods:

```
void addLink(IlvLinkImage obj, boolean redraw)
```

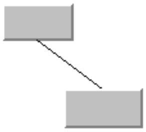
```
void addLink(IlvLinkImage obj, int layer, boolean redraw)
```

Here is a small piece of code that creates a grapher, two nodes and a link:

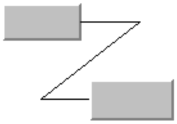
```
IlvGrapher grapher = new IlvGrapher();
IlvGraphic node1 = new IlvLabel(new IlvPoint(0,0), "node 1");
grapher.addNode(node1, false);
IlvGraphic node2 = new IlvLabel(new IlvPoint(100, 0), "node 2");
grapher.addNode(node2, false);
IlvLinkImage link = new IlvLinkImage(node1, node2, true);
grapher.addLink(link, false);
```

The library provides a set of predefined links:

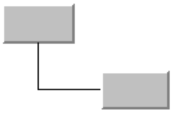
- `IlvLinkImage` - a direct link between two nodes.



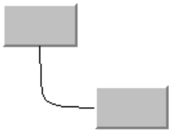
- `IlvPolylineLinkImage` - a link defined by a polyline.



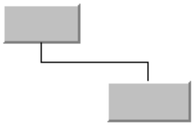
- `IlvOneLinkImage` - a link defined by two lines forming a right angle.



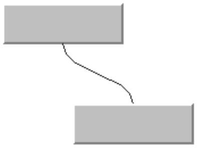
- `IlvOneSplineLinkImage` - a link that shows a spline.



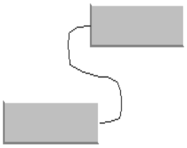
- `IlvDoubleLinkImage` - a link defined by three lines forming two right angles.



- `IlvDoubleSplineLinkImage` - a link defined by two spline angles.



- `IlvSplineLinkImage` - a free-form spline capable of creating mono- or multicurve links.



Contact Points

When a link is created between two nodes, it is attached to the default contact point of each node. Each node has five default contact points, one at the center of each side of its bounding rectangle and one at the center of the bounding rectangle. The contact point actually used depends on the location and size of the origin and destination node. The following are examples of connections:

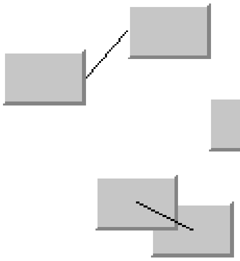


Figure 5.2 *Examples of Node Connections*

The grapher also provides a way to specify the contact points you need on a graphic object. This is done by using *link connectors*, which are subclasses of the class `IlvLinkConnector`,

Using Link Connectors

The class `IlvLinkConnector` is dedicated to the computation of the connection points of links. Subclasses of this abstract class can be used to obtain different contact points than the default ones. An `IlvLinkConnector` is associated with a node. The implementation of the method `getConnectionPoint` decides where the connection point of a link (provided as an argument) should be located.

An instance of `IlvLinkConnector` can be specified for each node of a grapher. To do this, simply create it using the constructor `IlvLinkConnector(IlvGraphic)` or use the method `attach(IlvGraphic, boolean)`. Notice that the same instance of link connector cannot be shared by several nodes.

A link connector specified for a node controls the connection points of all the links incident to this node. If you need the connection points of the incident links to not be all computed in the same way (that is, by the same link connector), you can specify a link connector individually for each extremity of each link. To do this, simply create it using the constructor `IlvLinkConnector(ilog.views.IlvLinkImage, boolean)` or use the method `attach(IlvLinkImage, boolean, boolean)`. Notice that the same link connector can be shared by several links incident to the same node.

To get the instance of link connector actually used to compute the contact point of a given link, use the static method `Get(IlvLinkImage, boolean)`.

Using the Class IlvPinLinkConnector

The subclass `IlvPinLinkConnector` manages the link connections to a node. Each link is attached to what we call a *pin*. An instance of the class `IlvPinLinkConnector` may be installed on a graphic object. This instance holds a set of pins. Each pin describes the position of a contact point on a node.

When creating an `IlvPinLinkConnector`, the instance is empty and does not contain any pins. You must provide a set of pins describing the position of the contact points that you need. The pins are defined by the class `IlvGrapherPin`. This class is an abstract class because its `getPosition` method is an abstract method. For this reason, you must first create a subclass of the `IlvGrapherPin` class. Specifying an implementation to the `getPosition` method enables you to indicate the position of the *grapher pin*. The signature of this method follows:

```
IlvPoint getPosition(IlvTransformer t)
```

The position of the pin depends on the transformer used to draw the node. This transformer is passed to the `getPosition` method. To compute the position of the pin, you may need to know the position of the node. For its position, use:

```
IlvGraphic getNode()
```

You may also decide to allow or inhibit the connection of a certain type of link to this pin. To do so, you will overwrite the `allow` method of your pin, which is called when you create a link with an interactor:

```
boolean allow(Object oClass, Object dClass,
              Object linkOrClass, boolean origin)
```

The interactor may highlight only the authorized pin based on the result of this method.

Once the pin classes are created, you will add the pin to the previously created instance of `IlvPinLinkConnector` using the following method:

```
void addPin(IlvGrapherPin pin)
```

Example: Defining Your Connection Points

This example defines two classes of pins:

- The class `InPin` that allows links going to the node
- The class `OutPin` that allows the links going from the node.

  The pins of class `InPin` are placed on the left border of the object and the pins of type `OutPin` on the right border.

The classes are:

```
final class InPin extends IlvGrapherPin
{
  static final int numberOfPins = 5;
  int index;

  public InPin(IlvPinLinkConnector connector, int index)
  {
    super(connector);
    this.index = index;
  }
  protected boolean allow(Object orig, Object dest,
                          Object linkOrClass,
                          boolean origin)
  {
    return !origin;
  }

  public IlvPoint getPosition(IlvTransformer t)
  {
    IlvRect bbox = getNode().boundingBox(null);
    IlvPoint p = new IlvPoint(bbox.x,
                              bbox.y+(bbox.height/(numberOfPins+1)*
                              (index+1));
    if (t != null) t.apply(p);
    return p;
  }
}
```

In this example, five instances of `InPin` will be created. Each pin has an index giving its position on the node. The `getPosition` method returns the position of the pin on the left side of the node according to its index. The `allow` method returns `true` only for links going to this pin (parameter `origin` is `false`). The `OutPin` class is very similar:

```
final class OutPin extends IlvGrapherPin
{
  static final int numberOfPins = 5;
  int index;

  public OutPin(IlvPinLinkConnector connector, int index)
  {
    super(connector);
    this.index = index;
  }

  protected boolean allow(Object orig, Object dest,
                          Object linkOrClass, boolean origin)
  {
    return origin;
  }

  public IlvPoint getPosition(IlvTransformer t)
  {
    IlvRect bbox = getNode().boundingBox(null);
    IlvPoint p = new IlvPoint(bbox.x+ bbox.width,
                              bbox.y+(bbox.height/
                                        (numberOfPins+1))*(index+1));
    if (t != null) t.apply(p);
    return p;
   }
}
```

The pins are located on the right side and only allow links leaving the node.

If `node` is a graphic object, the method that adds the pins to the node is:

```
grapher.addNode(node, 1, false);
IlvPinLinkConnector lc = new IlvPinLinkConnector(node);
for (int i = 0; i < 5; i++) {
  new InPin(lc, i);
  new OutPin(lc, i);
}
```

If you want to connect a link to a particular pin, use the method `connectLink` of the class `IlvPinLinkConnector`:

```
public void connectLink(IlvLinkImage link, IlvGrapherPin pin,
boolean origin)
```

Grapher Interactor Class

The library provides several *view interactors* to create links with the mouse.

The `IlvMakeLinkInteractor` is an interactor that allows a link of type `IlvLinkImage` to be created by selecting the origin and destination node.

This interactor can be customized so that it creates your own type of link. The link is created by the method `makePolyPoint`. This method uses the `getFrom` and `getTo` methods to determine the selected graphic objects:

```
protected IlvGraphic makePolyPoint(IlvPoint[] points)
{
  return new IlvLinkImage(getFrom(), getTo(), isOriented());
}
```

If you override this method, you must also override the method `getLinkClass` that returns the class of objects created by this interactor.

The class `IlvMakePolyLinkInteractor` is a subclass of the `IlvMakeLinkInteractor` class that allows you to create a link of class `IlvPolylineLinkImage`.

Creating a New Class of Link

In this section, you will learn how to create a new class of link. The link that is described here is a link defined by a polyline, whose starting and ending positions are fixed and are based on the starting and ending positions of the nodes. This link is a subclass of the `IlvLinkImage` class.

IlvPolylineLinkImage

Below is the beginning of the class:

```
public class IlvPolylineLinkImage extends IlvLinkImage
{
  private IlvPoint points[] = null;

  public IlvPolylineLinkImage(IlvGraphic from, IlvGraphic to,
                              boolean oriented, IlvPoint[] points)
  {
    super(from, to, oriented);
    init(points);
  ...
}
```

As you can see, we define a private field, `points`, that will contain all the intermediate points of the link. The origin and destination points are not contained in this array.

The constructor calls the corresponding constructor of `IlvLinkImage` and initializes the object. The `init` method simply fills the `points` field:

```
private void init(IlvPoint[] pts)
{
  if (pts == null)
    return;
  int i;
  points = new IlvPoint[pts.length];
  for (i = 0; i < pts.length ; i++)
    points[i] = new IlvPoint(pts[i].x, pts[i].y);
}
```

We also define a constructor that copies the object:

```
public IlvPolylineLinkImage(IlvPolylineLinkImage source)
{
  super(source);
  init(source.points);
}
```

getLinkPoints

The `getLinkPoints` method will now return the points defining the shape of the link. This method is used by `IlvLinkImage` to draw the object and to define the bounding rectangle of the object. In the class `IlvLinkImage`, this method only returns the origin and destination points of the link. For our polyline object, we use the `getLinkPoints` method to add the intermediate points of the link:

```
public IlvPoint[] getLinkPoints(IlvTransformer t)
{
  int nbpoints = getPointsCardinal();
  IlvPoint[] pts = new IlvPoint[nbpoints];
  if (nbpoints > 2)
    for (int i = 1 ; i < nbpoints-1; i++) {
      pts[i] = new IlvPoint(points[i-1]);
      if (t != null) t.apply(pts[i]);
    }
  pts[0] = new IlvPoint();
  pts[nbpoints-1] = new IlvPoint();
  getConnectionPoints(pts[0], pts[nbpoints-1], t);
  return pts;
}
```

For this, the `getConnectionPoints` method is used. The `getConnectionPoints` method computes the origin and destination point of the link. These points may depend on the connection pins on the origin or destination object.

getPointCardinal, getPointAt

These methods, originating from the interface `IlvPolyPointsInterface`, are defined like this:

```
public int getPointsCardinal()
{
  if (points == null)
    return 2;
  else
    return points.length +2;
}

public IlvPoint getPointAt(int index, IlvTransformer t)
{
  if (index == 0 || index == getPointsCardinal()-1)
  {
   IlvPoint[] pts = new IlvPoint[2];
   pts[0] = new IlvPoint();
   pts[1] = new IlvPoint();
   getConnectionPoints(pts[0], pts[1], t);
   return pts[(index == 0) ? 0 : 1];
  }
  else
  {
    IlvPoint p = new IlvPoint(points[index-1]);
    if (t != null)
      t.apply(p);
    return p;
  }
}
```

allowsPointInsertion, allowsPointRemoval

Since we want to be able to add and remove points from the editing interactor associated with links (that is, `IlvLinkImageEditInteractor`), we override the `allowPointAddition` and `allowPointRemoval` methods to return `true`:

```
public boolean allowsPointInsertion()
{
  return true;
}

public boolean allowsPointRemoval()
{
  return points != null && points.length >= 1;
}
```

Since the above methods return `true`, the `insertPoint` and `removePoint` methods will be called from the interactor. We define them like this:

```
public void insertPoint(int index, float x, float y,
                        IlvTransformer t)
{
  if (points == null && index == 1) {
    points = new IlvPoint[1];
    points[0] = new IlvPoint(x,y);
  }
  else if (index == 0)
    throw new IllegalArgumentException("bad index");
  else if (index >= getPointsCardinal())
    throw new IllegalArgumentException("bad index");
  else index --;
  if (index >= 0 && index <= points.length){
    IlvPoint[] oldp = points;
    points = new IlvPoint[oldp.length+1];
    System.arraycopy(oldp, index, points, index + 1,
                     oldp.length - index);
    points[index] = new IlvPoint(x,y);
    if (index > 0)
      System.arraycopy(oldp, 0, points, 0, index);
  }
  else throw new IllegalArgumentException("bad index");
}

public void removePoint(int index, IlvTransformer t)
{
  if (index ==0) return;
  if (index == getPointsCardinal()-1)
    return;
  index --;
  if (points != null && index >= 0 && index < points.length)
  {
    IlvPoint[] oldp = points;
    points = new IlvPoint[oldp.length-1];
    if (index > 0)
      System.arraycopy(oldp, 0, points, 0, index);
      int j = oldp.length - index - 1;
      if (j > 0)
      System.arraycopy(oldp, index + 1, points, index, j);
  }
  else throw new IllegalArgumentException("bad index");
}
```

applyTransform

The method `applyTransform` is called when the bounding box is to be modified (when the object is moved or enlarged, for example). We simply apply the transformation to the intermediate points:

```
public void applyTransform(IlvTransformer t)
{
  if (getPointsCardinal() > 2 && points != null)
    for (int i = 0 ; i < points.length; i++) {
     if (t != null) t.apply(points[i]);
    }
}
```

Input/Output

We can now add input/output methods to allow the saving and reading of the intermediate points:

The method `write`:

```
public void write(IlvOutputStream stream) throws IOException
{
  super.write(stream);
  stream.write("points", points);
}
```

The corresponding `IlvInputStream` constructor:

```
public IlvPolylineLinkImage(IlvInputStream stream) throws
                                              IlvReadFileException
{
  super(stream);
  IlvPoint[] points = stream.readPointArray("points");
  init(points);
}
```

CHAPTER

6

Nested Managers and Nested Graphers

This chapter introduces the notion of nested managers and graphers. Nesting allows you to add a manager in another manager or a grapher in another grapher.

The following topics are covered:

- *Introducing Submanagers*
- *Nested Managers*
- *Manager Frame*
- *Expanding and Collapsing*
- *Nested Graphers*
- *Selection in a Nested Manager*
- *Content-Changed Events in Nested Managers*
- *Interactors and Nested Managers and Graphers*

Introducing Submanagers

Chapter 4, *Managers* and Chapter 5, *Graphers* introduced the manager (`IlvManager`) and its subclass the grapher (`IlvGrapher`). The manager and the grapher are the main classes that can contain graphic objects for displaying and manipulating in several views.

The manager and grapher are themselves graphic objects that can be embedded inside another manager or grapher. This nesting feature of ILOG JViews allows you to create applications that display a graph inside another graph. An example is shown in Figure 6.1.

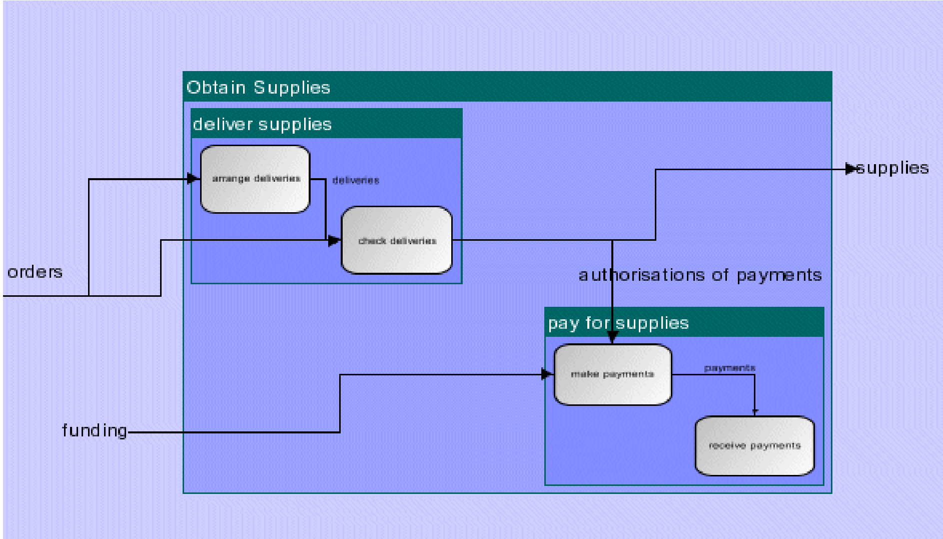


Figure 6.1 *Nested Graphs*

In this figure the object titled “Obtain Supplies” is a grapher (instance of `IlvGrapher`) that itself contains two other graphers, titled “deliver supplies” and “pay for supplies.” The figure shows that a manager (and a grapher) can be surrounded by a frame; in this example the three (blue) frames each display the name of the manager as a title. This type of decoration as well as the background of the submanager can be completely customized. Another feature shown in this example is the fact that links between nodes can cross subgraph boundaries. Such links are called *intergraph* links.

Each grapher or manager embedded inside another manager or grapher can also be displayed in several views, just like top-level managers.

Finally, a manager or grapher embedded inside another manager has two different representations: an expanded state where all the objects contained in the submanager are visible, and a collapsed state where the manager is drawn with a collapsed representation that can also be customized.

Nested Managers

The `IlvManager` class inherits from the `IlvGraphic` class; as a consequence, a manager and a grapher can be added in another manager or grapher just like any other graphic objects. To add a manager in a manager, you will simply use the `addObject` method of the `IlvManager` class.

Example: Adding a Nested Manager

Here is the code for a simple example:

```
import ilog.views.*;
import ilog.views.graphic.*;
import javax.swing.*;
import java.awt.*;

public class SubManagerExample
{
  public static void main(String[] args) {
    IlvGraphic obj;
    IlvManager toplevel = new IlvManager();
    IlvManager subManager = new IlvManager();

    obj = new IlRectangle(new IlvRect(10,10,50,50), false, true);
    subManager.addObject(obj, false);
    obj = new IlvRectangle(new IlvRect(100,100,50,50), false, true);
    subManager.addObject(obj, false);

    toplevel.addObject(subManager, false);

    obj = new IlvRectangle(new IlvRect(10,200,50,50), false, true);
    toplevel.addObject(obj, false);

    IlvManagerView view = new IlvManagerView(toplevel);
    view.setBackground(Color.blue);
    JFrame frame = new JFrame("Sub manager Example");
    frame.getContentPane().add(view);
    frame.setSize(200,200);
    frame.setVisible(true);
  }

}
```

This simple example creates two `IlvManager` objects, the top-level manager (variable `toplevel`) that will be displayed in the view and the submanager (variable `subManager`). The submanager is added in the top level by the line:

```
toplevel.addObject(subManager, false);
```

Two rectangles are also added to the submanager. Another rectangle is added at the top level.

The resulting application is shown in Figure 6.2.

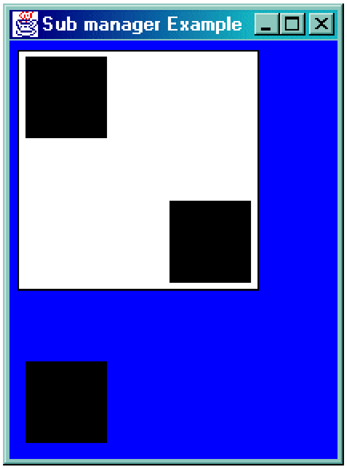


Figure 6.2 Submanager Example

The white area is the submanager containing two rectangles.

> ***Note:*** *Adding a manager to a manager can be done to an infinite level. The library will just make sure that you do not create cycles in the hierarchy of nested managers.*

Traversing Nested Managers

Once a manager has been added in a manager, it is no different from any other graphic object. Nevertheless, the manager contains a set of methods that allow you to have quick access to the managers added in a manager and to the manager hierarchy in general.

You can obtain an enumeration of the managers present in a submanager using the method:

```
IlvGraphicEnumeration getManagers()
```

and a count of how many managers are present with the method:

```
int getManagersCount()
```

The same methods can be applied to a particular layer:

```
IlvGraphicEnumeration getManagers(int layer)
```

```
int getManagersCount(int layer)
```

Note that these methods will return the submanagers at the first level of the hierarchy only. Using these methods is much more efficient than traversing the list of all objects present in a manager.

You can also traverse up the hierarchy of managers, using the method:

```
IlvManager getParent()
```

In our small example (see *Example: Adding a Nested Manager* on page 95) the following line would return `toplevel`:

```
subManager.getParent()
```

For a manager at the top level, this method returns `null`.

Coordinate System In Nested Managers

Note that in our small example (see *Example: Adding a Nested Manager* on page 95) the position and size of the submanager has not been specified. The position and size of a submanager depends on the position and size of the objects that are contained in the submanager. As a consequence, all objects that are contained in the submanager are always displayed, and the size and position of the submanager may change when a graphic object contained in this manager changes position or size.

As for any graphic object, you can move and resize a submanager using the `IlvManager.moveObject` or `IlvManager.reshapeObject` methods. In the example you could do something like:

```
toplevel.moveObject(subManager, 100,100, true)
```

When a nested manager is resized or moved, the objects contained in the manager do not change position in the submanager coordinate system. The submanager will move because an affine transformation (an `IlvTransformer`) will be specified in the submanager to define a new relative coordinate system for the submanager. In the line above for moving the `subManager` to `(100,100)`, the transformation is a simple translation of `(100,100)`.

To obtain the affine transform that defines the coordinate system of the submanager, you can use the method:

```
   IlvTransformer getTransformer()
```

If the submanager has not been moved or reshaped, this returns the identity transformation.

To know what transformation is used to draw a submanager in a specified view, you can use the following method of the `IlvManager` class:

```
   IlvTransformer getDrawingTransformer(IlvManagerView)
```

This method returns the affine transformation used to draw the objects in a manager. If the specified view is a view of the manager, then it simply returns the affine transform of the view; otherwise, the method will compose the transformation of all the parents of the manager and also the transformation of the view to give the result.

Working with Graphic Objects In Nested Managers

The API you will use to manipulate graphic objects stored in a nested manager is the same as the one you use when working on a top-level manager. For example, you will use the `moveObject` or `reshapeObject` method of the `IlvManager` class to move or reshape an object in a nested manager. You will also use the `applyToObject` method of `IlvManager` when modifying a property of a graphic object that changes the bounding box of a graphic object. The only difference for a nested manager is that when a graphic object is stored in a nested manager, changing its size or moving it can change the size of the manager itself, proceeding recursively up the hierarchy of the manager.

The `IlvManager` class provides some convenient methods for working with graphic objects in nested managers; these methods have a `traverse` parameter that when set to `true` means that the method applies also to nested managers in this manager.

You can access all the objects in the hierarchy of managers using the following methods with the `traverse` parameter set to `true`:

- To return the total number of objects in the hierarchy use the method:

  ```
  int getCardinal(boolean traverse)
  ```

- To return an enumeration of all objects use the method:

  ```
  IlvGraphicEnumeration getObjects(boolean traverse)
  ```

- You can locate an object under a certain point using:

  ```
  IlvGraphic getObject(IlvPoint p, IlvManagerView view,
        boolean traverse)
  ```

- Finally, you traverse the hierarchy of objects to apply a function:

  ```
  void map(IlvApplyObject f, Object arg, boolean traverse)
  ```

  This method applies the function `f` to all objects of the hierarchy when the `traverse` parameter is set to `true`.

  -
    ```
    void mapIntersects(IlvApplyObject f, Object arg,
          IlvRect rect, IlvTransformer t, boolean traverse)
    ```

    This method applies the function to all graphic objects that intersects the specified rectangle in the hierarchy of nested managers.

  -
    ```
    void mapInside(IlvApplyObject f, Object arg,
          IlvRect rect, IlvTransformer t, boolean traverse)
    ```

    This method applies the function to all graphic objects that are inside the specified rectangle in the hierarchy of the nested manager.

Some methods allow you to deal with selection and deselection of objects in a hierarchy of nested managers. This is explained in *Selection in a Nested Manager* on page 109.

View on a Nested Manager

Just as for a top-level manager, it is possible to view the contents of a nested manager in several views (`IlvManagerView`). Any modification of an object in a nested manager will be reflected in all the views in which the object appears, which are:

- All the views of the submanager that contain the object.
- All the views of the parent manager, and proceeding recursively.

The association of the view with a submanager is no different from the association of the view for the top-level manager.

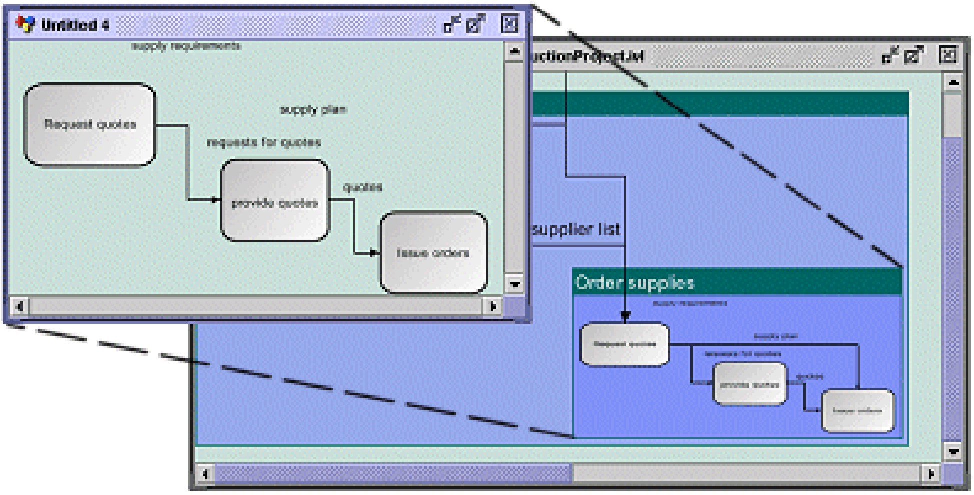


Figure 6.3 *Views on Nested Manager*

Manager Frame

When a manager is a nested manager, a frame can be drawn around the objects it contains.

Defining the Frame

The frame around a nested manager is defined by the interface `ilog.views.IlvManagerFrame`. A default frame is provided when creating a nested manager; the default frame is defined by the class `ilog.views.IlvDefaultManagerFrame`.

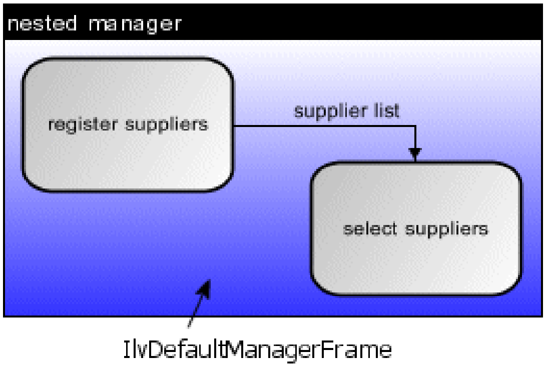


Figure 6.4 *Default Manager Frame*

This `IlvManagerFrame` interface defines the margin that will be added around the manager. It defines how the frame is drawn and how hit testing on the frame is performed.

To specify the frame that must be drawn around the manager, you use the following methods of the `IlvManager` class:

```
void setFrame(IlvManagerFrame frame)

IlvManagerFrame getFrame()
```

Note that you can remove the frame from the manager using the `setFrame` method with a `null` parameter.

Defining the Margins

The following methods of the `IlvManagerFrame` interface define the margins that are added around the manager:

```
float getBottomMargin(IlvManager manager, IlvTransformer t)

float getLeftMargin(IlvManager manager, IlvTransformer t)

float getRightMargin(IlvManager manager, IlvTransformer t)

float getTopMargin(IlvManager manager, IlvTransformer t)
```

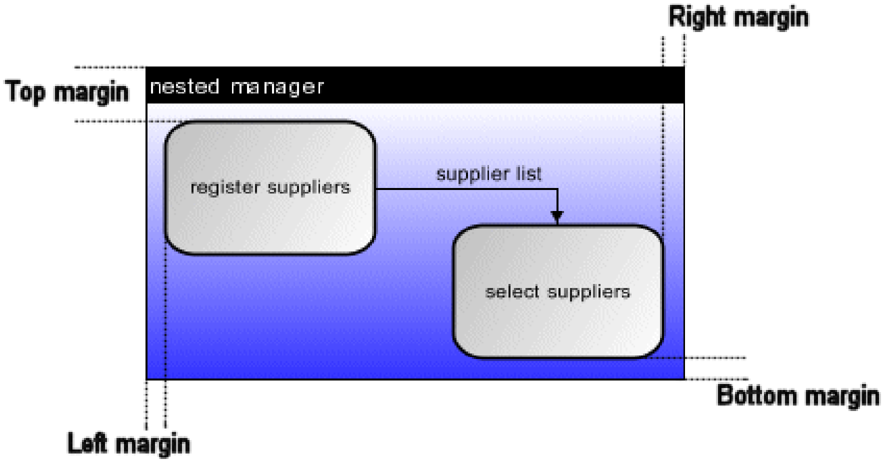


Figure 6.5 *Manager Frame Margins*

Drawing the Frame

The interface defines a method to draw the frame:

```
void draw(IlvManager manager, IlvRect bbox, Graphics g,
       IlvTransformer t)
```

This method already provides the manager, with its bounding box in the view coordinate system that already takes the margins into account. The `IlvDefaultManagerFrame` fills the background and draws a border around the manager. It also draws a title at the top of the manager that corresponds to the name of the manager, as you can see in the picture above (Figure 6.5).

If the frame implementation fills the background of the manager, the frame is an opaque frame and the following method must return `true`:

```
boolean isOpaque(IlvManager manager)
```

The `IlvDefaultManagerFrame` can be optionally opaque.

- If the frame is opaque, graphic objects under the nested manager are hidden. This disallows manipulating these graphic objects.
- If the frame is not opaque, graphic objects under the nested manager are visible. This allows manipulating the graphic objects whether covered by the frame or not.

Hit Testing

To allow hit testing on the frame, the interface defines the `contains` method:

```
boolean contains(IlvManager manager, IlvPoint p,
      IlvPoint tp, IlvTransformer t)
```

When the frame is not opaque (transparent), this method must return `true` for a possible border and not for the background.

Saving the Frame to an IVL File

To be able to save the frame to an IVL file, the implementation must be a public class that also implements the `ilog.views.io.IlvPersistentObject` interface.

Copying a Frame

To be able to copy a manager that uses a frame, the frame implementation must define the `copy` method defined by the interface that must create a clone of the frame:

```
IlvManagerFrame copy()
```

Expanding and Collapsing

A nested manager can be expanded or collapsed. When collapsed, a nested manager has a different representation, as illustrated in Figure 6.6.

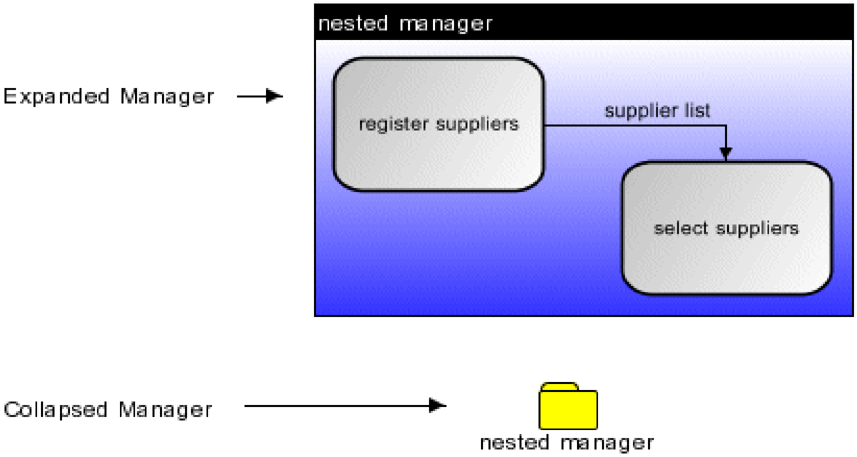


Figure 6.6 Expanded and Collapsed Manager

To expand/collapse a manager, the `IlvManager` class provides the following methods:

```
void setCollapsed(boolean collapse)
```

```
boolean isCollapsed()
```

When a nested manager is collapsed, the contents of the manager as well as the frame are no longer drawn. The manager has a new graphic representation, as you can see in Figure 6.6.

Defining the Collapsed Representation

The graphic representation that defines the collapsed manager is a graphic object, an instance of `ilog.views.IlvGraphic`. This allows you to define any kind of collapsed representation for your manager. The default graphic object used to draw a collapsed manager is an instance of the class: `ilog.views.graphic.IlvDefaultCollapsedGraphic`, which represents a folder above the name of the manager. To change this default representation, you use the following methods of the class `IlvManager`:

```
void setCollapsedGraphic(IlvGraphic graphic)
```

```
IlvGraphic getCollapsedGraphic()
```

Note that the graphic object used for the collapsed representation cannot be shared by several managers.

When collapsing, the collapsed graphic will be placed at the center of the area of the manager. When expanding, the manager will move so that its center will be placed at the center of the collapsed graphic.

Expand/Collapse Events

When the manager is expanded or collapsed, it fires an event to notify you of this change. The event is defined by the class `ilog.views.event.ManagerExpansionEvent`.

To listen to such events, you must create a listener that implements the `ilog.views.event.ManagerExpansionListener` interface, which defines the two methods:

- `void managerCollapsed(ManagerExpansionEvent evt)`

  which is called after the manager is collapsed.

- `void managerExpanded(ManagerExpansionEvent evt)`

  which is called before the manager is expanded.

You will then register your listener using the following method of the `IlvManager` class:

```
void addManagerExpansionListener(ManagerExpansionListener
listener)
```

Nested Graphers

As a subclass of a manager, a grapher (instance of `IlvGrapher`) inherits all the behavior of managers. A grapher can be nested in another manager or grapher. It can be collapsed or expanded, and a frame can be set on it. In addition to having the features inherited from the `IlvManager` class, nested graphers allow you to create applications that define graphs containing subgraphs, with links crossing the graph boundaries (intergraph links).

Intergraph Links

An intergraph link is a link that crosses grapher boundaries. In other words, an intergraph link is a link whose origin and destination are not in the same grapher. Some examples are shown in Figure 6.7.

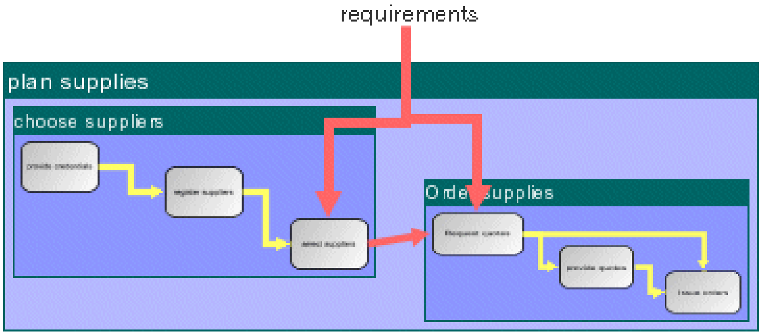


Figure 6.7 *Intergraph Links and Regular Links*

In this picture the red (darker) links are intergraph links and the yellow (light) links are regular links.

Creating an Intergraph Link

Given that the intergraph link has an origin and a destination in different graphers, the question is in which grapher the link itself is stored.

First of all, to be able to create intergraph links, you must have a hierarchy of nested graphers (`IlvGrapher`). The hierarchy should not contain a manager (`IlvManager`); otherwise, the library will not let you create an intergraph link. The intergraph link between a node stored in grapher A and a node stored in grapher B must be stored in the first common ancestor of A and B, as shown in Figure 6.8.

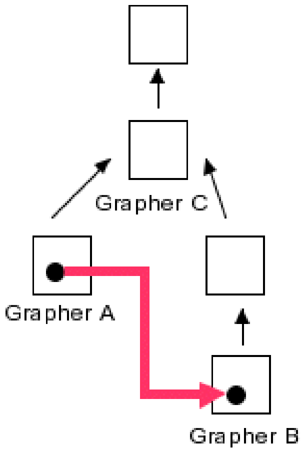


Figure 6.8 *Intergraph Link in a Hierarchy of Graphers*

In this picture the red intergraph link that connects an object of A and an object of B must be stored in grapher C, the first common ancestor of A and B.

The `IlvGrapher` class provides a static utility method that allows you to determine the first common grapher:

```
static IlvGrapher getLowestCommonGrapher(IlvGraphic obja,
      IlvGraphic objb)
```

To create an intergraph link, the code will look like this (assuming that the origin and destination variables have been created and added in different graphers):

```
IlvGraphic origin, destination;
IlvLinkImage link;

...

link = new IlvLinkImage(origin, destination, false);

IlvGrapher common = IlvGrapher.getLowestCommonGrapher(origin, destination);

common.addLink(link, false);
```

The `IlvGrapher` class also provides a static utility method that allows you to directly add the link in the common parent grapher:

```
static void addInterGraphLink(IlvLinkImage link, boolean redraw)
```

The code above is then equivalent to:

```
IlvGraphic origin, destination;
IlvLinkImage link;

...

link = new IlvLinkImage(origin, destination, false);

IlvGrapher.addInterGraphLink(link, false);
```

Accessing the Intergraph Link

The `IlvGrapher` class also provides methods that let you access the intergraph links stored in a grapher in an efficient way:

```
IlvGraphicEnumeration getInterGraphLinks()
```

```
int getInterGraphLinksCount()
```

Since an intergraph link is stored like other links in the grapher, such links are also part of the list of all the objects of the grapher returned by the `getObjects` method of the class `IlvManager`:

```
IlvGraphicEnumeration getObjects()
```

Nevertheless, the `getInterGraphLinks` method is much more efficient than traversing all objects of the grapher.

To distinguish an intergraph link from other objects in the grapher, you can use the following method of the `IlvGrapher` class:

```
boolean isInterGraphLink(IlvGraphic obj)
```

This method returns `true` if the specified graphic object is a link stored in this grapher with the origin or the destination stored somewhere else, in other words, if the graphic object is an intergraph link.

The `IlvGrapher` class also gives you access to the intergraph links that are leaving or entering a grapher. You can access such links using the methods:

```
IlvGraphicEnumeration getExternalInterGraphLinks()
```

```
int getExternalInterGraphLinksCount()
```

The difference between the methods `getInterGraphLinks` and `getExternalInterGraphLinks` is that the first method returns the intergraph links stored in this grapher with an origin or destination in another grapher, and the second method returns the intergraph links stored in another grapher but with the origin or destination in this grapher.

Figure 6.9 shows an example to illustrate this:

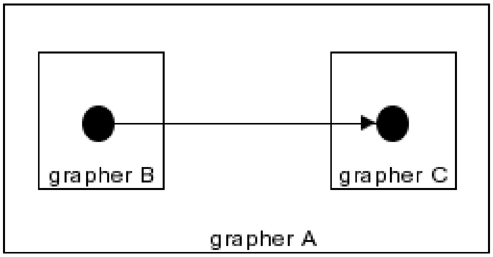


Figure 6.9 *External Intergraph Link*

Grapher A contains two graphers, B and C. The intergraph link from an object of B to an object of C is then stored in A. This link is an intergraph link of A (returned in `getInterGraphLinks` called on A) and is also an external intergraph link of B and C (returned by `getExternalInterGraphLinks` called on B or C).

Coordinate System of Intergraph Links

Since an intergraph link has its origin and destination in different graphers, it may be difficult to determine in which coordinate system the bend points of an intergraph link are defined.

The coordinate system of the link and its bend points is always the coordinate system of the grapher to which the link has been added. The only difference compared to regular links is the way an intergraph link computes its end points: the connection points of the link to its origin and destination. The link itself, in fact the base class of links (`ilog.views.IlvLinkImage`), computes its correct connection points, using the method:

```
void getConnectionPoints(IlvPoint src, IlvPoint dst,
      IlvTransformer t)
```

that stores the result in the `src` and `dst` parameters. If the link is an intergraph link, this method will compute the connection points in the coordinate system of the origin and destination nodes.

First the `getConnectionPoints` method determines if a link connector (`ilog.views.IlvLinkConnector`) is installed on the destination or origin, using the method:

```
boolean getLinkConnectorConnectionPoint(boolean origin,
      IlvPoint p, IlvTransformer t)
```

In this method the origin parameter is `true` for computing the connection point at the origin of the link and `false` for the destination of the link, and the transformer is the transformer used to draw the link.

This method returns `true` if a link connector is installed. If no link connector is installed, a default connection point is computed. If a link connector is installed, the link connector then computes the connection point with the method:

```
IlvPoint getConnectionPoint(IlvLinkImage link,
          boolean origin, IlvTransformer t)
```

- When the link is a regular link, the transformer is the transformer used to draw the link, which is the same as the transformer used to draw the origin and destination.
- When the link is an intergraph link, then the transformer is the transformer used to draw the origin or destination.

So when developing a new class of links or link connectors, there is no special work to be done to take into account the specific case of intergraph links.

Collapsed Grapher and Intergraph Links

When a grapher is collapsed, the graphic objects that it contains are no longer visible on the screen. The nodes that are the destination and origin of an intergraph link are not visible on the screen, and thus such intergraph links cannot visually point to their end nodes. These intergraph links will point to the collapsed representation of the grapher on the screen.

Here is an example of intergraph links in a grapher that is expanded (Figure 6.10) and collapsed (Figure 6.11):

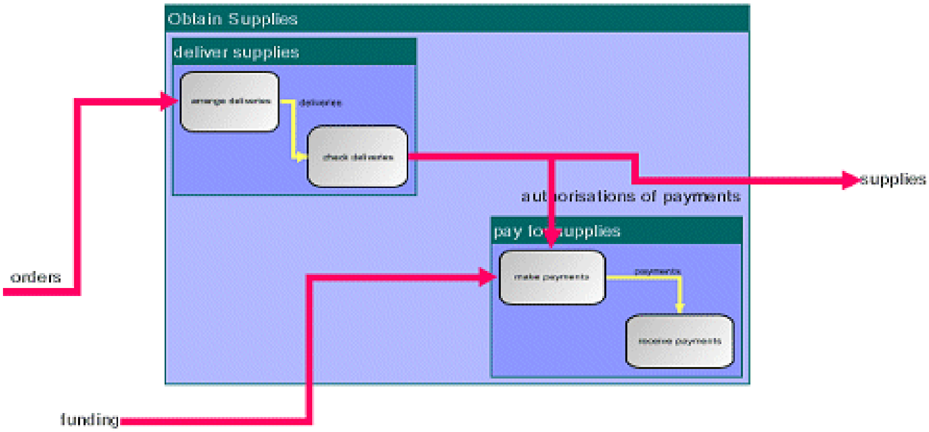


Figure 6.10 *Intergraph links to an Expanded Grapher*

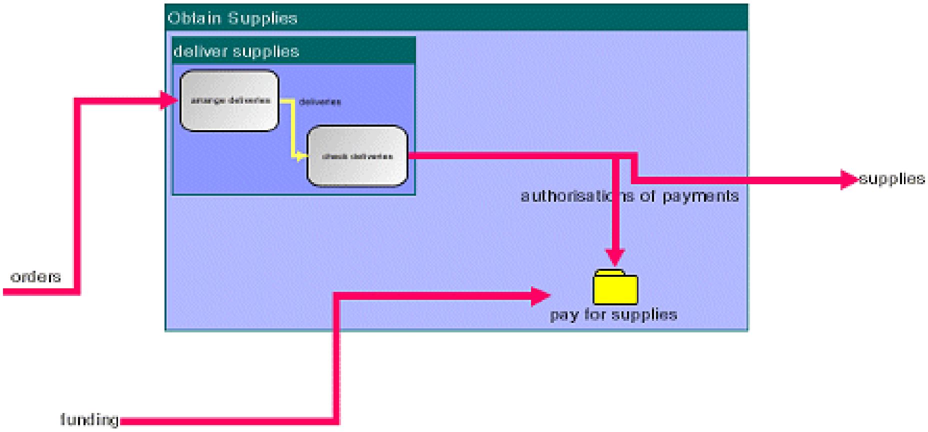


Figure 6.11 *Intergraph Links to a Collapsed Grapher*

Although the links are visually pointing to the collapsed representation of the manager once the manager is collapsed, the real origin and destination of the links do not change. The methods `getFrom` and `getTo` of the link (`ilog.views.IlvLinkImage`) are still returning the same object. Only the graphical representation changes in this case. The link visually points to the first noncollapsed parent manager.

In this case the position of the connection point of the link is then determined by the link connector installed on the collapsed manager and not by the ones installed on the real end nodes.

Creating a Link Using IlvMakeLinkInteractor

We have seen how to create an intergraph link by code. It is also possible to create intergraph links (and also regular links) using the `IlvMakeLinkInteractor`. When this interactor is installed, it allows you to interactively create a link from any graphic object to any other graphic object. This allows you to create intergraph links and also links to or from a nested grapher.

Selection in a Nested Manager

An `IlvManager` allows you to select objects that it contains as explained in *Selection in a Manager* on page 64. When a manager contains other managers, objects located in a nested manager can be selected using the same method of `IlvManager`:

```
void setSelected(IlvGraphic obj, boolean select, boolean redraw)
```

The `IlvManager` class provides the methods that allow you to query the selection status of objects in a hierarchy of nested managers and also to listen to selection events in such a hierarchy of managers.

Selection Methods for Nested Managers

Here is a set of methods that allows you to deal with the specific case of selections in a hierarchy of nested managers.

```
void selectAll(boolean traverse, boolean redraw)
```

This method selects all the objects in the manager and also all the objects in nested managers when the traverse parameter is `true`.

```
void deSelectAll(boolean traverse,boolean redraw)
```

This method deselects all the selected objects in the manager and also all the selected objects in nested managers when the traverse parameter is `true`.

```
IlvGraphicEnumeration getSelectedObjects(boolean traverse)
```

This method returns an enumeration that contains all the selected objects in this manager and in nested managers if the traverse parameter is set to true.

```
int getSelectedObjectsCount(boolean traverse)
```

This method returns the number of selected objects of this manager and in the nested managers if the traverse parameter is set to `true`.

```
void deleteSelections(boolean redraw, boolean traverse,
      boolean redraw)
```

This method removes the selected objects in the manager and also removes the selected objects in nested managers when the traverse parameter is set to `true`.

```
IlvSelection getSelection(IlvPoint p, IlvManagerView view,
      boolean traverse)
```

Finally, this method returns the selection object under the specified point. The method will search for selection objects in nested managers if the traverse parameter is set to `true`.

Selection Events

When a graphic object is selected or deselected, the manager will fire a selection event. This is described in *Listener on the Selections in a Manager* on page 67.

When a selection listener is registered in the manager using the method:

```
void addManagerSelectionListener(ManagerSelectionListener
listener)
```

it will only receive selection events for selections and deselections that are occurring in the manager where the listener was registered. To listen to selections that are taking place throughout a hierarchy of nested managers, the `IlvManager` class provides the following methods:

```
void addManagerTreeSelectionListener(
      ManagerSelectionListener listener)
```

```
void removeManagerTreeSelectionListener(
      ManagerSelectionListener listener)
```

When you register a selection listener using these methods, whenever an object is selected or deselected from a manager that is a submanager of the hierarchy of this manager, the listener will receive the event. Such a listener placed at the top-level manager of a hierarchy will then receive all the selection events of the hierarchy. To distinguish which submanager has sent the event, you can then use the method `getManager` on the event; the event is an instance of the class `ManagerSelectionChangedEvent` and contains the method:

```
IlvManager getManager()
```

Selection Interactor

The selection interactor (the class `ilog.views.interactor.IlvSelectInteractor`) allows you to select and edit objects in a hierarchy of nested managers. Using this interactor, you can select objects in several managers that are part of the hierarchy of nested managers.

Selecting Multiple Objects

You can select several objects either by:

- Using Shift-Click on objects , or
- Dragging a rectangle. When dragging a rectangle, graphic objects that are inside the rectangle will be part of the selection even if the submanager that contains these objects is not fully inside the rectangle.

Moving a Nested Manager

Clicking and dragging on the background of a nested manager will allow you to move the nested manager. To start a multiple selection with a selection rectangle, you can either:

- Click and drag in the background of the view, or
- Click and drag in the background of a nested manager with the Control (Ctrl) key pressed. In this case the Control key allows you to distinguish between beginning the drag of the selection rectangle and beginning the move operation of a manager.

Content-Changed Events in Nested Managers

When the content of the manager changes, for example, when an object is added or removed or when the bounding box of an object changes, the manager will fire a `ManagerContentChangedEvent` event. Any class can listen for the modification of the content of the manager by implementing the `ManagerContentChangedListener` interface. This mechanism is described in *Listener on the Content of the Manager* on page 62.

Registering such a listener in a manager using the `addManagerContentChangedListener` method of the manager allows the listener to receive Content Changed events only for modifications that are taking place in the manager where the listener is registered. A listener registered using this method will not be notified, for example, when a new graphic object is added in a submanager.

> ***Note:*** *Such a listener can nevertheless receive events indirectly due to some modifications in submanagers. For example, when a new graphic object is added in a submanager B of a manager A, the nested manager B may change size, so a listener registered on A may receive an* `ObjectBBoxChangedEvent` *due to the insertion of a new graphic object in B.*

In order to receive all Content Changed events of a hierarchy of nested managers, the `IlvManager` class allows you to register a global listener, with the methods:

```
void addManagerTreeContentChangedListener(
      ManagerContentChangedListener listener)
```

```
void removeManagerTreeContentChangedListener(
      ManagerContentChangedListener listener)
```

Such a listener registered at the top-level manager of a hierarchy will receive all the Content Changed events of the hierarchy. To distinguish which submanager has sent the event, you can then use the method `getManager` on the event; the event is an instance of the class `ManagerContentChangedEvent` and contains the method:

```
IlvManager getManager()
```

> ***Note:*** *You can register a global Content Changed listener on the top-level manager to detect all insertions of submanagers in the hierarchy of managers. If you do this, you must take into account the possibility that a manager that already contains some managers might be added to a manager. In this case the listener will not receive an event for each manager.*

Interactors and Nested Managers and Graphers

All interactors that are part of the ILOG JViews library and that work with individual graphic objects can work with objects embedded in a nested manager or grapher. We have seen that it is possible to:

- Select and edit objects in nested managers using the `IlvSelectInteractor` (see *Selection Interactor* on page 111).
- Create links in nested managers as well as creating intergraph links using the `IlvMakeLinkInteractor` (see *Creating a Link Using IlvMakeLinkInteractor* on page 109).

We discuss here the case of interactors in a nested manager or grapher that allow you to create new objects and also interactors set on graphic objects: `IlvObjectInteractor`.

Creation Interactors

The same interactors that allow you to interactively create a graphic object allow you to directly create objects in a submanager. Such interactors are following the same schema: When the interaction (the first mouse click) starts in a nested manager, then the created object will be placed in this manager. This is the case for all subclasses of `IlvMakeRectangleInteractor` and `IlvMakePolyPointsInteractor`, which are the base classes for creating an object with a rectangular shape or a set of points. It is also the case for the `IlvEditLabelInteractor` that allows you to create labels.

Object Interactors

Object interactors (instances of `IlvObjectInteractor`) are objects that describe the interaction for a specific graphic object. Such interactors are described in *Object Interactors* on page 68.

An object interactor that is set on a graphic object contained in a hierarchy of nested manager will still receive events coming from the view attached to a parent manager. There is no difference in writing an object interactor that can be used on such a graphic object.

CHAPTER

7

ILOG JViews Graphics Framework Printing

This chapter describes the printing framework used by the ILOG JViews Graphics Framework Component.

Introduction

The ILOG JViews Component Suite provides a generic printing framework described in the *ILOG JViews Printing Framework User's Manual*. The ILOG JViews Graphics Framework provides extensions of this generic printing framework that allow you to print structured graphics (an `IlvManager` object) in multiple pages or in a flow of text.

Since this chapter explains how the generic printing framework is extended for the purpose of vector graphics, we recommend that you first read the manual describing the generic printing framework.

The Graphics Framework printing classes are located in the `ilog.views.print` package.

The architecture of this package is based on four main components:

- A document that defines the printing configuration.
- The `IlvManager` to print.
- **The print UI:** some specialized dialog boxes and interactors that allow you to configure printing properties such as the page format, the orientation, the header and footer, the number of pages, and so forth.
- **The printing controller**: A high-level class that manages the document to print. You can use it to invoke the Print dialog box or the Page Setup dialog box, to preview the document, and to send the document to the selected printer.

We can distinguish three ways to print the contents of an `IlvManager` in a document:

- *Printing a Manager in Multiple Pages*

  You can print a manager (or an area of a manager) in multiple pages. Using the predefined document class (`IlvManagerPrintableDocument`) you simply have to specify the manager object you want to print and some printing parameters such as the number of pages and the area to print.

- *Printing a Manager in a Flow of Text*

  You can print a manager (or an area of a manager) in a flow of text.

- *Printing a Manager in a Custom Document*

  You can print a manager (or an area of a manager) in a custom document structure. In this mode you create your own document and insert a printable object that represents a manager in a page.

Printing a Manager in Multiple Pages

The `ilog.views.print` package contains several classes that allow you to easily print the contents of an `IlvManager` in multiple pages.

IlvManagerPrintableDocument

The Graphics Framework provides a document class, `IlvManagerPrintableDocument,` which is a subclass of the generic `IlvPrintableDocument` class. The `IlvManagerPrintableDocument` class is dedicated to printing the contents of a manager in multiple pages.

Unlike the `IlvPrintableDocument` class, you do not have to create pages and add them in the document. The `IlvManagerPrintableDocument` class will create the pages for you, depending on the parameters you specify for the document.

In addition to the generic parameters defined in the superclass `IlvPrintableDocument` such as the name and author of the document, the page format, the header and footer, and the page order, the `IlvManagerPrintableDocument` class allows you to specify the following options:

- The number of pages.
- The area of the manager to be printed.
- The zoom level used for printing.

The following code creates an `IlvManagerPrintableDocument` to print the area (0,0,500,500) of a manager in five columns:

```
IlvManagerPrintableDocument document = new IlvPrintableManagerDocument
                    ("My Document", view);
document.setColumnCount(5);
document.setPrintArea(new IlvRect(0,0,500,500));
```

Number of Pages

The number of pages is determined by the number of rows and columns that you specify.

- If you specify the number of rows, the document computes the number of columns necessary to cover the area to print.
- If you specify both the number of rows and the number of columns, then the document class will choose to use the number of rows or the number of columns to produce the minimum number of pages.

To print the manager in one page, you can set the number of rows and the number of columns to 1.

Area to Print

The area of the manager to print is specified by the `setPrintArea` and `getPrintArea` methods. When no print area has been specified, then the printed area will be the full area of the manager. To reset the area to print to the full area of the manager, you can simply call:

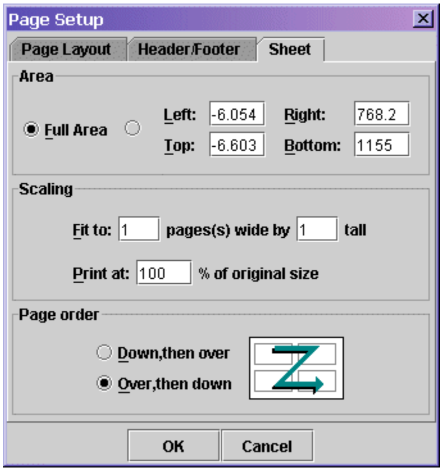
```
document.setPrintArea(null);
```

Zoom Level for Printing

The contents of the manager may be graphically different when a different zoom level is used, in particular when the manager contains nonzoomable objects. Thus, when printing the manager you may need to specify the zoom level used for printing. By default the contents of the manager are printed using the identity affine transform (that is, the zoom level 1).

IlvManagerDocumentSetupDialog

All properties of the `IlvManagerPrintableDocument` can be specified by a dialog box (class `IlvManagerDocumentSetupDialog`). This dialog box, a subclass of the generic `IlvDocumentSetupDialog`, contains an additional page in the tabbed pane that allows you to specify the area to print, the number of columns and rows (that is, the number of pages), the zoom level at which to print, and the page order for the numbering of pages.



Area

● Full Area ○ Left: -6.054 Right: 768.2 Top: -6.603 Bottom: 1155

Scaling

Fit to: 1 pages(s) wide by 1 tall

Print at: 100 % of original size

Page order

○ Down,then over

● Over,then down

You may not want to allow the user to change the zoom level, or you may need to specify a range of zoom level that is allowed for this specific manager. To do this you will use the following methods in the `IlvManagerPrintableDocument` class:

- Enable or disable the modification of the zoom level from the dialog box:

  `is/setZoomLevelModificationEnabled()`

- Set the minimum or maximum zoom level that can be used for printing.

  `setMaximimumZoomLevel(double zl)`

  `void setMinimumZoomLevel(double zl)`

Note that you do not have to create this dialog box yourself; the printing controller will manage an instance of this class for you.

IlvManagerPrintingController

The `IlvManagerPrintingController` is a subclass of the generic `IlvPrintingController` that controls the printing of an `IlvManagerPrintableDocument`.

After creating your `IlvManagerPrintableDocument`, you must to create an `IlvManagerPrintingController` for the document. Then you will be able to perform various actions on the document to be printed; for example, you can:

- Call the `printDialog` method to invoke a dialog box to select a printer or to specify other printer-related information.
- Call the `setupDialog` method to open the Page Setup dialog box.
- Call the `printPreview` method to preview your document.
- Call the `print` method of the print controller to print your document. You can also click the print button while you preview the document.

IlvManagerPrintAreaInteractor

The `ilog.views.print` package also contains a specific interactor (`IlvManagerPrintAreaInteractor`) that allows you to specify the area to print on the manager itself by dragging a rectangle.

Example

The example below is a full application using an `IlvManagerPrintingController` and `IlvManagerPrintableDocument` to print the contents of a manager in multiple pages.

This example shows a Swing frame containing an `IlvManagerView` that displays the contents of a manager. This application has also a menu bar with standard printing menu items such as `Print...`, `Print Preview`, `Page Setup...`, and so forth.

```
import java.awt.*;
import java.awt.print.*;
import java.awt.event.*;
import javax.swing.*;
import ilog.views.print.*;
import ilog.views.util.print.*;
import ilog.views.*;
import ilog.views.interactor.*;
import ilog.views.swing.*;

/**
 * This is a very simple example to show how to use
 * the IlvManagerPrintableDocument class.
 */
public class ExamplePrint extends JFrame
{
  /**
   * The manager to print.
   */
  IlvManager manager;

  /**
   * An IlvManagerView to display the content of a manager.
   */
  IlvManagerView mgrview;

  /**
   * The printing controller.
   */
  IlvManagerPrintingController controller;

  /**
   * The interactor that allows you to specify the area to print.
   */
  IlvPrintAreaInteractor printAreaInteractor;

  /**
   * Creates and initializes the example.
   */
  public ExamplePrint() {

    super("Printing Example");
    getContentPane().setLayout(new BorderLayout());

    // Creates the manager to print.
    manager = new IlvManager();

    // Fills the manager with some data.
    try {
       manager.read("data.ivl");
    } catch (Exception e) {
    }


    // Creates a view of the manager.
    mgrview  = new IlvManagerView(manager);
    mgrview.setBackground(Color.white);
    mgrview.setKeepingAspectRatio(true);

    // Creates the printing controller
    controller = new IlvManagerPrintingController(mgrview) ;

    // Initializes the document with some parameters.
    IlvPrintableDocument document = controller.getDocument();

    document.setName("data.ivl");
    document.setAuthor("My name");

    // Creates the interactor.
    printAreaInteractor = new IlvPrintAreaInteractor(controller);

    // Adds a toolbar to edit/zoom/pan.
    IlvJManagerViewControlBar toolbar
          = new IlvJManagerViewControlBar();
    toolbar.setView(mgrview);

    // Creates a scroll manager view.
    IlvJScrollManagerView scrollview
             = new IlvJScrollManagerView(mgrview);
    getContentPane().add(scrollview, BorderLayout.CENTER);
    getContentPane().add(toolbar, BorderLayout.NORTH);
    scrollview.setPreferredSize(new Dimension(300,300));
    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    addWindowListener(new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
          System.exit(0);
        }
    });

    // Creates the menu bar.
    setJMenuBar(createMenu());

  }

  private JMenuBar createMenu() {

    JMenuBar bar = new JMenuBar();
    JMenu file = new JMenu("File");
    JMenu parea = new JMenu("Print Area");
    JMenuItem preview = new JMenuItem("Print Preview...");
    JMenuItem setup = new JMenuItem("Page Setup...");
    JMenuItem setarea = new JMenuItem("Set Print Area");
    JMenuItem cleararea = new JMenuItem("Clear Print Area");
    JMenuItem print = new JMenuItem("Print...");


    // Action to open the print preview dialog.
    preview.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          controller.printPreview(ExamplePrint.this);
        }
      });

    // Action to open the setup dialog box.
    setup.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          controller.setupDialog(ExamplePrint.this, true, true);

        }
      });

    // Action to print the document.
    print.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          try {
            controller.print(true);
          } catch (Exception e) { }

        }
      });

    // Action to install the print area interactor.
    setarea.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          mgrview.setInteractor(printAreaInteractor);
        }
      });


    // Action to reset the print area to full manager size.
    cleararea.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          ((IlvManagerPrintableDocument)controller.
                getDocument()).setPrintArea(null);
        }
      });


    file.add(setup);
    file.add(parea);
    parea.add(setarea);
    parea.add(cleararea);
    file.add(preview);
    file.add(print);

    bar.add(file);

    return bar;

  }
  public static void main(String[] args)
  {
    ExamplePrint example = new ExamplePrint();
    example.pack();
    example.setVisible(true);

  }
}
```

Printing a Manager in a Flow of Text

The `IlvManagerPrintableDocument` class allows you to print the contents of a manager. You may also need to add a view of a manager in a flow of text. To do this you use an instance of the `IlvPrintableDocument` class and the `IlvFlow` class described in the *ILOG JViews Printing Framework User's Manual*.

The `IlvFlow` class not only allows you to control the flow of text in a textual document, but it also allows you to insert images in the text. The objects you can insert in the text are implemented by the interface `IlvFlowObject`.

The `ilog.views.print` package provides an implementation of the `IlvFlowObject` interface for printing the manager. The class is named `IlvSimplePrintableManager`.

Printing a Manager in a Custom Document

The ILOG JViews Component Suite printing framework also allows you to create your own document structure by creating pages (`IlvPage` class) and adding printable objects in those pages (`IlvPrintableObject` class). For more information see the *ILOG JViews Printing Framework User's Manual*.

The `ilog.views.print` package also provides a new printable object (subclass of `IlvPrintableObject`) that you can insert in a page of your document to print an area of a manager. This class is named `IlvPrintableManagerArea`.

CHAPTER

8

Composer, the ILOG JViews Editor

This section describes how to use Composer, the ILOG JViews editor. The following topics are covered:

- *Introducing the ILOG JViews Editor*
- *Running Composer*
- *Object-Editing Buffers*
- *Creating Objects*
- *Creating a Network of Graphic Objects*
- *Using the Graph Layout Algorithms*
- *Options for Creating Objects*
- *Editing Modes*
- *Inspecting and Editing Objects in the Properties Sheet*
- *Editing Objects*
- *Controlling Buffers*
- *Printing and Saving Files*
- *Setting Files and Startup Options*
- *Exiting Composer*

Introducing the ILOG JViews Editor

Composer lets you create and edit ILOG JViews graphic objects and generate ILOG JViews formatted files (`.ivl` files). The graphic objects are created and edited in object-editing buffers. These buffers represent a manager and its manager view. The objects of the buffer are stored and saved along with their coordinates within the manager.

The source code of Composer is written using ILOG JViews and Swing and is open to your modifications, such as the integration of your own *interactors* or *graphic objects*.

Running Composer

Before running Composer, an environment variable should be set:

- On UNIX: the `bin` directory of your JDK must be added to the `PATH` environment variable.
- On Windows: you should set the `JAVA_HOME` environment variable to the JDK installation directory.

To run Composer, first move to the `composer` directory at the following path:

```
<install>/bin/composer/
```

To launch Composer:

- On UNIX: use the `composer` shell script in the `composer` directory.
- On Windows: use the `composer.bat` file.

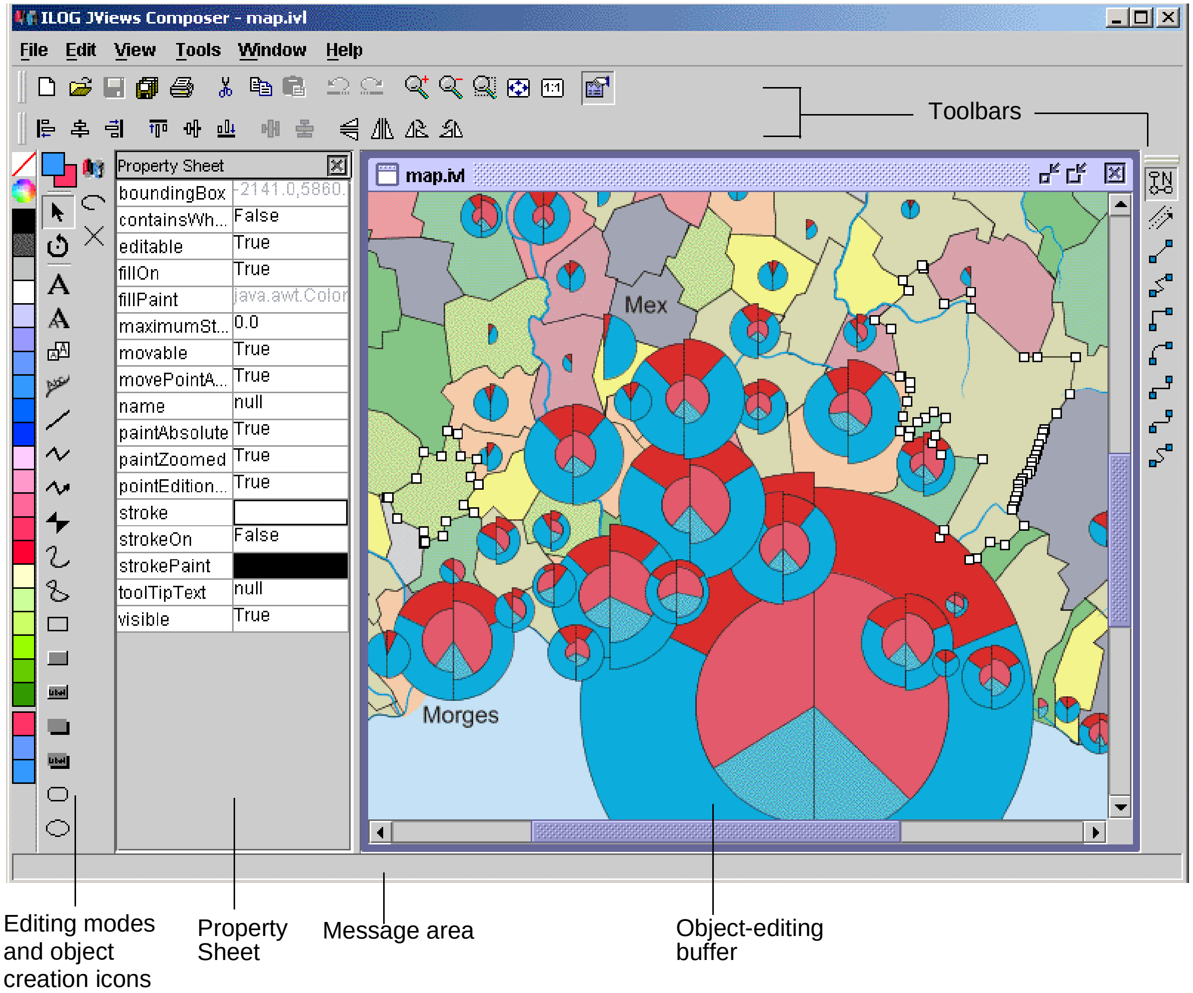
Composer has a special version to allow you to create domain-specific graphic objects (Prototypes). To launch this version, use the `-p` option:

```
>composer -p
```

The source code of Composer is located in:

```
<install>/bin/composer/src
```

When launching Composer, the main window is displayed:



Property Sheet

boundingBox	-2141.0,5860.
containsWh...	False
editable	True
fillOn	True
fillPaint	java.awt.Color
maximumSt...	0.0
movable	True
movePointA...	True
name	null
paintAbsolute	True
paintZoomed	True
pointEdition...	True
stroke	
strokeOn	False
strokePaint	■
toolTipText	null
visible	True

Toolbars

Editing modes and object creation icons

Property Sheet

Message area

Object-editing buffer

Figure 8.1 Composer Main Window

Object-Editing Buffers

By default, an empty object-editing buffer is displayed when Composer is launched. It is ready for you to add objects and begin editing them. It corresponds to an `IlvManager` object in the ILOG JViews API.

Creating Buffers

To create a new buffer, choose the New command from the File menu. Once a buffer is created, you can show an additional view on that buffer manager by choosing the Additional View command from the File menu. When the main view is closed, the additional view will be closed automatically.

If you have nested managers in your buffer, you can open a view on them by selecting one of them and choosing the Nested Manager View command from the contextual menu.

Loading a File to an Editing Buffer

If you need to load an existing file in a buffer, Composer allows you to load ILOG JViews formatted files (`.ivl`) and Scalable Vector Graphics (SVG) files (`.svg, .svgz`).

To open a file of one of these formats, choose the Open command from the File menu or the Open (📂) icon in the top toolbar.

Loading an ILOG JViews File

The ILOG JViews formatted file (`.ivl`) is the default format. Generally, you do not have to choose a different type in the “Files of type” combo box to load `.ivl` files. For `.ivl` files that contain your own graphic objects, the path to the graphic object classes must be present in the `CLASSPATH` variable when you launch Composer.

Loading an SVG File

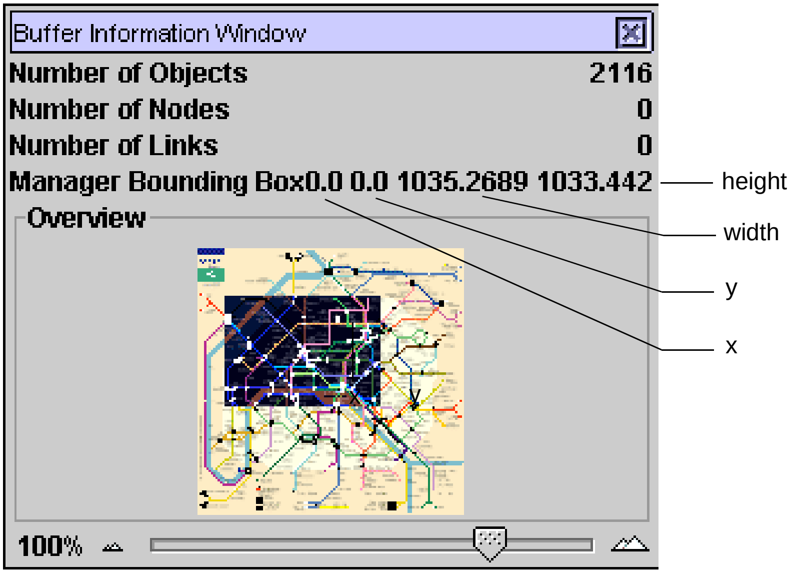
To open an SVG file (`.svg` format), choose “Scalable Vector Graphics Files” in the “Files of type” combo box. Some options for SVG file loading are available in the Tools > Options > SVG Configuration menu. See Table 11.4, *Supported SVG Elements*, to check which SVG elements are recognized by ILOG JViews.

Buffer Information Window

The Buffer Information Window, illustrated in Figure 8.2, displays the following information concerning the active buffer:

- **Number of Objects:** Number of objects in the manager and its possible nested managers.
- **Number of Nodes:** Number of nodes in the manager and its possible nested managers.

- **Number of Links:** Number of links in the manager and its possible nested managers.
- **Manager Bounding Box:** The manager bounding box is the smallest possible horizontal rectangle surrounding one or more objects in a manager. The first two numbers are the values of the x and y axes of the upper left corner of the manager bounding box (according to the manager coordinates). The last two numbers are its width and height.



Buffer Information Window

Number of Objects 2116

Number of Nodes 0

Number of Links 0

Manager Bounding Box0.0 0.0 1035.2689 1033.442

Overview

100%

Figure 8.2 Buffer Information Window

Below this information window you will find an overview of the buffer, which allows you to navigate through it and zoom in or out using the scale slider.

Displaying the Buffer Information Window

To display the Buffer Information Window:

1. Make the buffer whose Information Window you want to display active.
2. Choose Buffer Information Window from the View menu.

Docking the Buffer Information Window

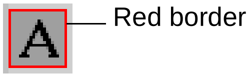
The Information Window as well as other Windows of the View menu can be docked for ease of use. To do this:

1. Drag the Information Window to one of the sides of the Composer main window.
2. When the window becomes transparent, release the mouse button. It becomes a frame in the main window.

Creating Objects

Table 8.1 lists the interactor icons on the left toolbar that allow you to create objects.

Once an object has been created, Composer automatically reverts to selection mode. To prevent Composer from reverting to selection mode so that you can create several of the same kinds of objects in a row without this interruption, double-click the icon of the object rather than single-clicking it. You will then be able to draw as many of these objects as you like until you select your next editing mode. When you double-click the icon, it takes on a red border to signal that the double-click has taken effect. For example, the label icon appears as follows:



To create the objects, use the interactor icons as described in Table 8.1.

Table 8.1 Object Creation Icons

A	To create a **label** (`IlvLabel`), click the icon and then click in the object-editing buffer to open the entry field. You may also edit a labelled object in this mode: click the label object icon and then click the label object in the buffer. This will open the editing field of the label.
A	To create a **zoomable label** (`IlvZoomableLabel`), click the icon and then click in the buffer.
abc	To create a **text on path** (`IlvTextPath`), click the icon and then click an existing `IlvGeneralPath`; the text will be created on this path. You can also click in the buffer instead of clicking an existing path; this will create a **text on path** with a default path.
╱	To create a **line** object (`IlvLine`), click the icon and then click two points in the buffer.
∿	To create a **polyline** object (`IlvPolyline`), click the icon and then click in the buffer to specify the control points and double-click when finished.
∿	To create a **polyline object with an arrow** (`IlvArrowPolyline`), click the icon and then click in the buffer to specify the control points and double-click when finished.
	To create a **polygon** object (`IlvPolygon`), click the icon and then click in the buffer to specify the control points and double-click when finished.

Table 8.1 Object Creation Icons (Continued)

∿	To create a **spline** (`IlvSpline`), click the icon and then click in the buffer to specify the control points and double-click when finished.
∿	To create a **closed spline** (`IlvClosedSpline`), click the icon and then click in the buffer to specify the control points and double-click when finished.
▭	To create a **rectangle** object (`IlvRectangle`), click the icon and then drag a rectangle in the buffer.
▬	To create a **relief rectangle** object (`IlvReliefRectangle`), click the icon and then drag a rectangle in the buffer.
Lbl	To create a **relief label** (`IlvReliefLabel`), click the icon and then click in the buffer to open the entry field.
▬	To create a **shadow rectangle** object (`IlvShadowRectangle`), click the icon and then drag a rectangle in the buffer.
Lbl	To create a **shadow label** (`IlvShadowLabel`), click the icon and then click in the buffer to open the entry field.
▢	To create a **round rectangle** object (`IlvRectangle` with non-`null` radius), click the icon and then drag a rectangle in the buffer.
⬭	To create an **ellipse** object (`IlvEllipse`), click the icon and then drag a rectangle in the buffer.
ILOG	To create an **icon** object (`IlvIcon`), click the icon and then drag a rectangle in the buffer. If you click without dragging a rectangle, you will be asked for the name of the **icon**, and it will be created using its default size.
✕	To create a **marker** object (`IlvMarker`), click the icon and then click in the buffer.
◠	To create an **arc** object (`IlvArc`), click the icon and then click in the buffer.

Most of the objects that can be created by dragging a rectangle can also be created with a default size by just clicking on the view without dragging a rectangle.

If during the creation process you want to cancel your action, you can press the ESC key to cancel the interactor.

Creating a Network of Graphic Objects

Composer allows you to create a network of graphic objects, which consists of nodes. All of the graphic objects presented in Table 8.1 on page 130 may be used as nodes that can then be linked together into a network using the node link icons on the Grapher Toolbar.

The Grapher Toolbar

To display the Grapher Toolbar, choose Toolbars from the View menu and then choose Grapher Toolbar. When displaying the Grapher Toolbar, the default node creation mode (the first icon on the left) is set. Table 8.2 lists the different link object interactors and their icons.

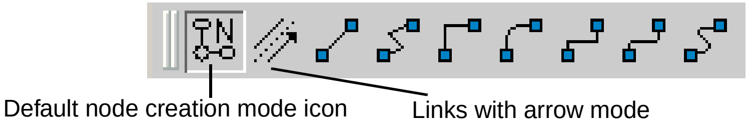


Figure 8.3 *Grapher Toolbar*

To create the links, use the interactor icons as described in Table 8.2.

Table 8.2 *Link Creation Icons*

	Creates an `IlvLinkImage` object (a straight link).
	Creates an `IlvPolylineLinkImage` (a link drawn as a polyline).
	Creates an `IlvOneLinkImage` (a link formed by two lines at a right angle).
	Creates an `IlvOneSplineLinkImage`.
	Creates an `IlvDoubleLinkImage` (a link formed by three lines at two right angles).
	Creates an `IlvDoubleSplineLinkImage`.
	Creates an `IlvSplineLinkImage`.

You can create the same objects with arrows by selecting the arrow mode in the Grapher Toolbar.

Creating Networks of Nodes

All objects created from the left toolbar are potential network nodes by default. To create a network of these objects, they only have to be linked. For this, you use the link object icons on the Grapher Toolbar. When you double-click a link object icon, noticeable by the red border created around it, you are able to create as many links as you want without having to click the icon again.

To create the links:

1. Display the link icons by choosing Toolbars from the View menu and then choosing Grapher Toolbar.
2. Click the icon corresponding to the type of link you want to create.
3. Click the origin node and then the destination node in the buffer.

Selecting All Nodes or Links

You can select all the nodes or links by choosing the Select all Nodes or the Select all Links command from the Edit menu.

Converting Objects into Nodes

If at some point you created objects while not in the default node-creation mode (because the node-creation mode icon was deselected), you can still have these objects converted into nodes in order to create a network with them. To convert the objects into nodes:

1. Select the objects you want to convert into nodes. (You may want to use the Select All command from the Edit menu.)
2. Choose the Convert to Node command from the Edit menu.

Using the Graph Layout Algorithms

Composer allows you to use the graph layout algorithms provided by ILOG JViews to easily obtain clear and readable representations of graphs and networks.

The Graph Layout Toolbar

The Graph Layout toolbar contains icons that allow you to select a graph layout algorithm, to apply the selected algorithm, and to inspect its parameters. To display the Graph Layout toolbar, choose Toolbars from the View menu and then choose Grapher Toolbar. You can also use the commands from the Graph Layout submenu of Tools. Table 8.3 describes the different Graph Layout icons on the Grapher toolbar.

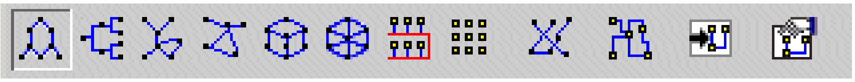


Figure 8.4 *Graph Layout Toolbar*

To select and apply a layout algorithm, use the interactor icons as described in the table.

Table 8.3 *Graph Layout Icons*

	Selects the `IlvHierarchicalLayout` algorithm.
	Selects the `IlvTreeLayout` algorithm.
	Selects the `IlvUniformLengthEdgesLayout` algorithm.
	Selects the `IlvSpringEmbedderLayout` algorithm.
	Selects the `IlvTopologicalMeshLayout` algorithm.
	Selects the `IlvCircularLayout` algorithm.
	Selects the `IlvBusLayout` algorithm.
	Selects the `IlvGridLayout` algorithm.
	Selects the `IlvRandomLayout` algorithm.
	Selects the `IlvLinkLayout` algorithm.
	Applies the selected layout algorithm to the graph in the active buffer.
	Shows the Graph Layout Inspector.

Selecting a Graph Layout Algorithm

The ILOG JViews Graph Layout module provides Graph Layout algorithms covering a wide range of graph types and representation standards. The *ILOG JViews Graph Layout User's Manual* contains detailed information about each algorithm and provides practical tips on how to choose the appropriate algorithm.

To select a graph layout algorithm for the current buffer, simply click its icon on the Graph Layout toolbar or use its corresponding command from the Graph Layout submenu of Tools. When you select a graph layout algorithm, you are able to customize the parameters using the Graph Layout Inspector. This is discussed in *Customizing a Graph Layout Algorithm* on page 135. Selecting a graph layout algorithm does not perform the algorithm. This is explained in *Performing the Graph Layout Algorithms* on page 138.

> ***Note:*** *When you select a layout algorithm, it applies to the current buffer. Changing buffers changes the layout algorithm to the one applied to the new current buffer.*

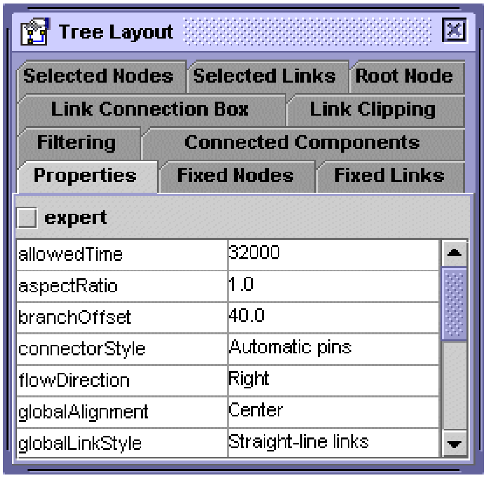
Customizing a Graph Layout Algorithm

Each graph layout algorithm has a set of parameters. The parameters have default values that are often convenient, but you may still need to customize these parameters or to choose other options for the layout process.

Editing with the Graph Layout Inspector

To edit the parameters of the selected graph layout algorithm, open the Graph Layout Inspector by clicking on its toolbar icon or by choosing the Graph Layout command from the Tools menu and then choosing Parameters.

The Graph Layout Inspector contains a Properties page and several pages that accompany it:



allowedTime	32000
aspectRatio	1.0
branchOffset	40.0
connectorStyle	Automatic pins
flowDirection	Right
globalAlignment	Center
globalLinkStyle	Straight-line links

Figure 8.5 *Graph Layout Inspector*

- **Properties page**—Provides a property sheet that is very similar to the Graphic Object property sheet. Its use is quite similar to the description you can find in *Inspecting and Editing Objects in the Properties Sheet* on page 149. In addition to the normal properties, you may choose to inspect expert properties with the check box that is provided, the expert setting displaying both normal and expert properties at the same time.

> ***Note:*** *Within the Graph Layout Inspector, Composer allows you to modify the parameters of a single graph layout algorithm in a different way for each buffer.*

- **Accompanying pages**—The pages that accompany the Properties page vary depending on the graph layout algorithm that is selected. These pages give you access to other parameters that cannot be edited using the property sheet.

Saving Parameter Settings

When you change layout parameters in the Properties page or the accompanying pages and later try to save the buffer, you can choose whether you want to save the buffer with or without the layout parameter settings. If you select the command:

- **Save As** from the File menu, a file selection dialog box opens which allows you to select the file type and the file name. The file type:
  - "`ILOG JViews Files (*.ivl)`" can be used to save the buffer without layout parameters. When you load the saved file later, all layout parameter settings will be lost.
  - "`ILOG JViews Files With Layout Parameters (*.ivl)`" can be used to save the buffer with layout parameters, so that the parameter settings are not lost.

- **Save** from the File menu, Composer saves the buffer:
  - With layout parameters if it was loaded with layout parameters.
  - Without layout parameters if it was loaded without layout parameters and no parameter changed.
  - In the format that you select from a confirmation dialog, if the buffer was loaded without layout parameters but some layout parameters have changed.

If the file is saved with layout parameter settings, the type of the last selected layout, considered as being the preferred layout for the graph, is also saved.

Loading Parameter Settings

When loading an `IVL` file, Composer determines automatically whether the file contains layout parameter settings.

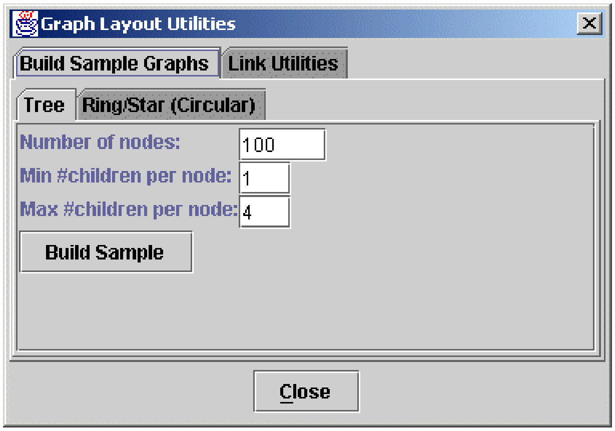
When loading an `IVL` file, Composer automatically loads the layout parameter settings stored in the file. If the preferred layout has been stored in the file, the corresponding icon in the layout toolbar and the corresponding item in the layout menu are automatically selected.

Creating Graphs/Networks

To see what graphs/networks can be created, you can experiment with the different graph layout algorithms using the sample `.ivl` files that are provided at `<installdir>/data/graphlayout`. There is a subdirectory for each graph layout algorithm containing typical graph examples. Each file is stored with information about the preferred layout to be executed for it. When the file is loading, the preferred layout is automatically selected in the layout toolbar and in the layout menu.

You can also build the graph from scratch using the graph editing capabilities of Composer (covered in the section *Creating a Network of Graphic Objects* on page 132), or load a sample file and modify it.

Lastly, you can automatically generate examples for certain types of graphs. This is done on the Build Sample Graphs pages in the Graph Layout Utilities window. To access these pages, choose Graph Layout in the Tools menu and then select Utilities:



Number of nodes: 100
Min #children per node: 1
Max #children per node: 4

Figure 8.6 Build Sample Graphs Pages

Two basic types of sample graphs can be created. This is done on the Tree or the Ring/Star (Circular) page. To create a sample, set the parameters that are proposed on its page and then click the Build Sample button. The sample graph is generated in a new, separate buffer.

Performing the Graph Layout Algorithms

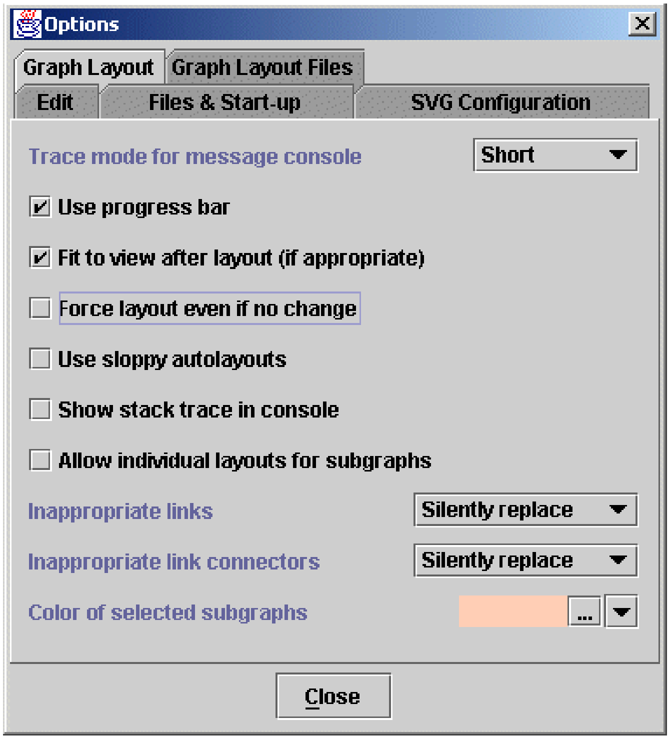
Once a graph has been constructed or loaded, you can apply the currently selected graph layout algorithm by clicking on its icon on the Grapher Toolbar, or by choosing Graph Layout on the Tools menu and then selecting Perform Layout.

Viewing Status Messages

By default, information about the result of the layout algorithm is displayed in the Message Console window. To access the Message Console, select Message Console from the View menu and then select Message Console.

Specifying Preferences for the Layout Process

You can specify preferences for the layout process on the Graph Layout page. To access this page, choose Options from the Tools menu and then select the Graph Layout tab:



Trace mode for message console: Short

☑ Use progress bar

☑ Fit to view after layout (if appropriate)

□ Force layout even if no change

□ Use sloppy autolayouts

□ Show stack trace in console

□ Allow individual layouts for subgraphs

Inappropriate links: Silently replace

Inappropriate link connectors: Silently replace

Color of selected subgraphs

Figure 8.7 *Graph Layout Page*

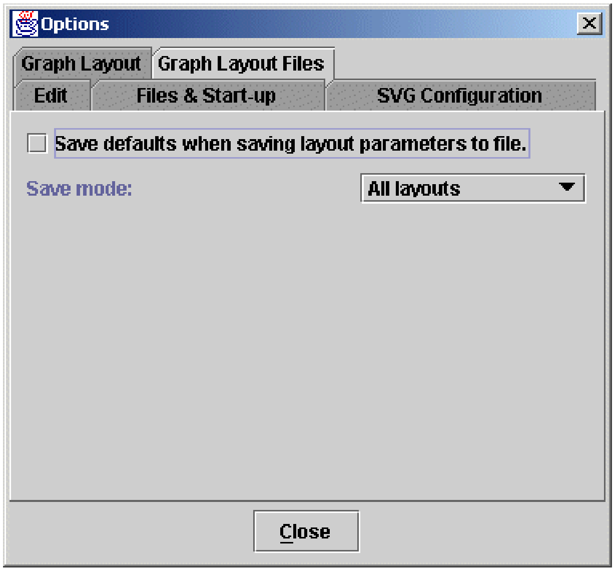
The following preferences can be set:

- **Trace mode for message console:** To establish the degree of detail you want in the message console messages.
- **Use progress bar:** Displays the progress bar during the layout process. Enabling the progress bar slightly slows down the process but provides ergonomical feedback.
- **Fit to view after layout (if appropriate):** A full view of the graph layout is performed on the layout display (except for the Link layout).
- **Force layout even if no change:** Forces layout to be redone even if the graph has not changed since the last time the layout was performed. Useful in testing and debugging situations.
- **Use sloppy autolayouts:** If enabled, a sloppy re-layout mechanism is used for layouts in autolayout mode instead of the standard autolayout mechanism.
- **Show stack trace in console:** Displays the stack trace of exceptions that occur outside the code of the layout algorithms.

- **Allow individual layouts for subgraphs:** If enabled, allows you to select individual layout styles for subgraphs and to inspect their parameters. For details, see *Laying Out Nested Graphs* on page 141.
- **Inappropriate links**: Allows you to choose the way Composer reacts when inappropriate links are found during layout.
- **Inappropriate link connectors**: Allows you to choose the way Composer reacts when inappropriate link connectors are found during layout.
- **Color of selected subgraphs:** Allows you to edit the background color of subgraphs selected by double-clicking for editing of the layout parameters.

Specifying Options for the Save Process

You can also specify options for the saving process using the Graph Layout Files page of the Options window. To access this window, choose Options from the Tools menu.



□ Save defaults when saving layout parameters to file.

Save mode: All layouts

Figure 8.8 *Graph Layout Files Page*

The following preferences can be set:

- **Save defaults when saving layout parameters to file:** Enables or disables saving layout parameters that have default values when saving graphs into IVL files with layout parameters.
- **Save mode:** Allows you to choose the layouts for which the parameters are saved.

Laying Out Nested Graphs

The ILOG JViews Graph Layout library supports advanced features for the recursive layout of nested graphs. For details on nested graphs, see Chapter 6, *Nested Managers and Nested Graphers*. For details on the layout of nested graphs, see the corresponding section in the Advanced chapter of the *ILOG JViews Graph Layout User's Manual*.

The recursive layout of nested graphs is available in Composer. When loading a file containing a nested graph, or when building a nested graph in Composer, you can press the Perform Layout button to lay out the entire nested graph; the layout applies the currently selected layout algorithm in a recursive way to the topmost grapher and its subgraphers.

However, for ergonomical reasons, the editing of the layout algorithms for subgraphs is not enabled by default in Composer. If you want to be able to select a different layout for different subgraphs in the hierarchy of graphs, or to make different settings, you need to enable the editing of the layout algorithms for the subgraphs. To do so, choose Options from the Tools menu, select the Graph Layout tab, and then check the Allow individual layouts for subgraphs check box.

When editing of layouts for subgraphs is enabled, it becomes possible to inspect the currently selected layouts for the subgraphs and to specify the layout algorithm to be applied to the subgraph.

Inspecting the Layout for the Subgraph

When the editing of layouts for subgraphs is enabled, the currently selected layout of a subgraph can be inspected in the standard way, as described in *Editing with the Graph Layout Inspector* on page 135. To indicate the subgraph you want to inspect, double-click the subgraph. Holding the shift key during the click allows you to select multiple subgraphs at the same time. You can deselect all subgraphs by a double-click in the white space of the top-level graph. If no subgraph is selected using the double-click, editing of layout parameters affects only the top-level graph.

If the Graph Layout Inspector is already open, it automatically updates its contents each time you double-click on subgraphs. If more than one subgraph is selected by double-clicking, the Graph Layout Inspector allows editing of the layout properties that are common to all the layouts currently selected for these subgraphs.

For comparison, when the editing of layouts for subgraphs is disabled, the Graph Layout Inspector always inspects the currently selected layout for the topmost grapher, which is applied recursively for nested graphs.

Choosing the Layout for the Subgraph

When individual layouts for subgraphs are enabled, the layout algorithm to be applied to each subgraph can be chosen in the standard way, as described in *Selecting a Graph Layout Algorithm* on page 135. When selecting a subgraph by double-click, the new layout chosen in the toolbar or in the menu becomes the new preferred layout for the selected subgraph.

The next time the layout is performed, the preferred layout individually chosen for each subgraph is applied. Note that the layout that is selected in the toolbar and in the menu automatically updates when you select or deselect subgraphs by double-click.

In this mode, to change the selected layout for the topmost grapher, you first deselect all subgraphs by a double-click in the white space of the toplevel graph, then you select the layout in the toolbar or in the menu. If you do not specifically select a different layout for a subgrapher, the same layout as for the topmost grapher is applied, with the same parameters. However, you can specify different settings for the layout of a subgrapher, even if it is the same layout algorithm as for the topmost grapher.

For comparison, when editing of the layouts for subgraphs is disabled, changing the selected layout in the toolbar or in the menu has the effect of always changing the layouts of all the graphers (topmost and all subgraphers). In this mode, the double-click on subgraphs has no effect.

Displaying Graph Statistics

The Statistics command from the Graph Layout submenu of the Tools menu calculates statistics on the graph and shows the results in the Message Console. It shows:

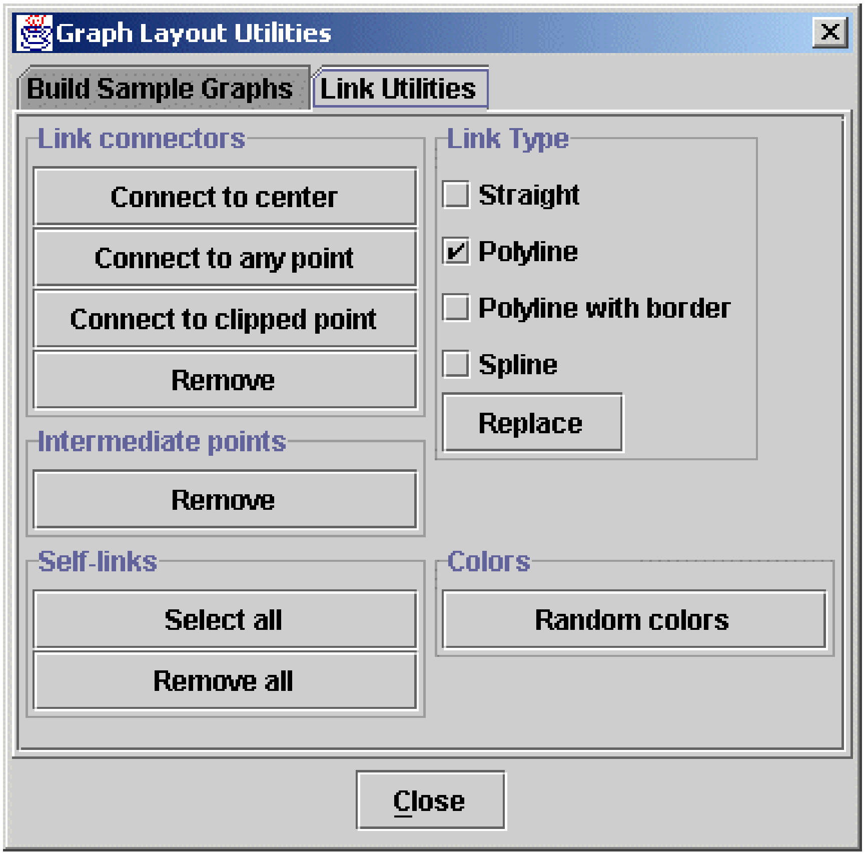
- The number of nodes of links that are filtered by a layout filter.
- The number of nodes and links that are unfiltered.
- The number of overlappings between unfiltered nodes.
- The number of overlappings between an unfiltered node and an unfiltered link.
- The number of overlappings between unfiltered link segments. Link segments overlap if they are parallel and have common points.
- The number of crossings between unfiltered links. Link segments cross if they are not parallel and have common points.
- The number of bends in unfiltered links.
- Whether the graph is connected or not.

Selecting Link Functionalities

The Link Utilities page of the Graph Layout Utilities window furnishes link functionalities.

> ***Note:*** *For detailed information on link connections, refer to Contact Points on page 84.*

To access the Link Utilities page, choose Graph Layout in the Tools menu and then select Utilities. In the Graph Layout Utilities window that appears, click the Link Utilities tab. The Link Utilities page is displayed:



☐ Straight
☑ Polyline
☐ Polyline with border
☐ Spline

Figure 8.9 *The Link Utilities Page*

Five sections to the page exist: Link connectors, Intermediate points, Link Type, Colors, and Self-links. Their functionalities, described below, all act on the selected nodes or links in the active buffer:

◆ **Link connectors**

- **Connect to center**: Installs link connectors such that the connection points are located at the center of the nodes. The package and class for this link connector is `ilog.views.IlvCenterLinkConnector`.
- **Connect to any point**: Installs link connectors such that the connection points can be located at any specified location. The package and class for this link connector is `ilog.views.graphlayout.IlvRelativeLinkConnector`.
- **Connect to clipped point**: Installs link connectors such that the connection points can be located at any specified location while being clipped by the border of the node. The

package and class for this link connector is `ilog.views.graphlayout.IlvClipLinkConnector`.

  - **Remove**: Removes all link connectors from the selected nodes.

- **Intermediate points**
  - **Remove**: Removes intermediate points from the selected links. Note that the following links cannot have their intermediate points removed: `IlvOneLinkImage`, `IlvOneSplineLinkImage`, `IlvDoubleLinkImage`, `IlvDoubleSplineLinkImage`.

- **Link Type**
  - **Straight**: Choice of the straight-line link type (class `IlvLinkImage`).
  - **Polyline**: Choice of the polyline link type (class `IlvPolylineLinkImage`).
  - **Polyline with border**: Choice of the polyline with border link type (class `BorderPolylineLinkImage` defined in Composer).
  - **Spline**: Choice of the spline link type (class `IlvSplineLinkImage`).
  - **Replace**: Replaces the selected links in the active buffer with the selected type of links.

- **Self-links**
  - **Select all**: Selects all self-links.
  - **Remove all**: Removes all self-links.

- **Colors**
  - **Random colors**: Gives a random color to each link. Along with using thicker lines for the links, random colors make it easier to differentiate the paths between the various links.

Options for Creating Objects

The following options allow you to specify the properties of your objects before you create them:

- *Aligning Objects on a Grid*
- *Specifying Layers for a Buffer*
- *Using Different Colors*
- *Using Different Fonts*
- *Changing the Ratio of Rectangle Objects*

Aligning Objects on a Grid

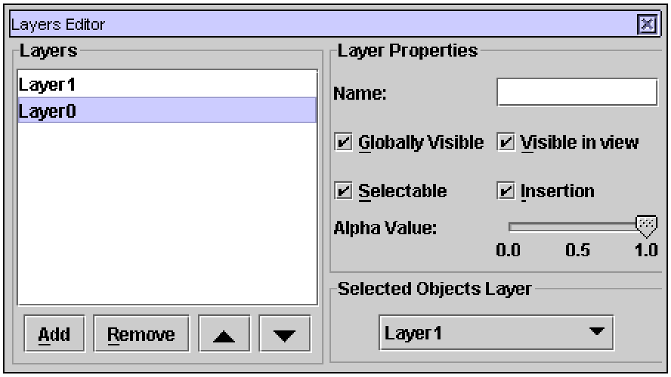
A magnetic grid can be installed on views in order to facilitate the alignment of objects as you create them. To display the grid, choose the Grid command from the Tools menu. The properties that can be set for the grid are as follows:

- **Visible:** To make the grid visible.
- **Active:** To make the grid active.
- **Horizontal Spacing:** To adjust the horizontal spacing of the grid points.
- **Vertical Spacing:** To adjust the vertical spacing of the grid points.
- **x, y:** To set the origin point of the grid, by default point (0, 0) in manager coordinates.

Specifying Layers for a Buffer

You place graphic objects into different *layers*, which are indexed. Objects placed in the layer of index n will be on top of objects of layer n-1. There are two layers created by default, Layer0 and Layer1. All objects created from the toolbar are placed in the insertion Layer0 by default.

To create new layers, edit layers, as well as to set certain properties pertaining to them, you use the Layers Editor window. To display this window, choose Layers Editor from the View.



Layer1
Layer0

Name:

☑ Globally Visible ☑ Visible in view

☑ Selectable ☑ Insertion

Alpha Value: 0.0 0.5 1.0

Layer1

Figure 8.10 Layers Editor Window

The different elements of the Layers Editor window are:

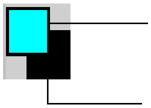
- **Layers:** The Layers pane lists the layers that exist in the buffer.
- **Add:** Clicking on the Add button adds a layer to the buffer. You can then select this newly created layer and click the Insertion toggle button in order to have the next objects you create inserted into this layer.

- **Remove:** The Remove button removes a selected layer from the buffer. (To select a layer, click it in the Layers pane.)
- **Layer Properties Name field:** By default, the layers are named as follows: Layer0, Layer1, Layer2, and so on. You can change the default name for something more descriptive and then use it in your code as well.
- **Globally Visible:** Specifies that the layer is visible in all views of the manager.
- **Visible in View:** Specifies that the layer is visible in the buffer view.
- **Selectable:** Specifies whether the objects in the layer can be selected or not.
- **Insertion:** By default, the objects are placed in layer 0. When you select a layer and click the Insertion toggle button, the newly created objects are placed in this layer.
- **Alpha Value:** Adjusts the transparency of a layer.
- **Selected Objects Layer:** Displays the layer of selected objects and allows you to edit their layers.

By default, the Layers Editor displays the layers of the top-level manager of the current buffer. If you want to display the layers of a nested manager of the current buffer, you can open a view on the nested manager (see *Creating Buffers* on page 128); the Layers Editor will then display the layers of the nested manager.

Using Different Colors

When you create objects, they are drawn with a gray background and a black border by default. If you want to create objects of another color, use the Choose Foreground and Background Color windows. To access these windows, click the Foreground or Background portion of the color icon on the left toolbar:



The foreground color: used for text, shadow, borders, and so on.

The background color: used for the fill-in or pattern color of the object, as well as for the graphic portion of a text object.

Changing this value will change the default colors used to create the objects. In addition to that, this will also change the colors of the selected objects in the active buffer.

You can also choose a color from the Color Bar that is displayed on the left of the Composer window. By clicking a color with the:

- Left mouse button you will choose it for the background color of the objects.
- Right mouse button you will choose it for the foreground color.

On the top of the Color Bar you will find a special color case for removing the background or border of objects. By clicking that color case with the:

- Left mouse button, the selected objects and the objects you create will no longer have a background.
- Right mouse button, the selected objects and the objects you create will no longer have borders.

> ***Note:*** *To edit the color properties of an existing object without changing the default values, see Editing Properties that Require a Special Editor on page 149.*

Using Different Fonts

The text of `IlvFontInterface` text objects, also have default properties. To create or edit text objects with another font or font size, use the Font Toolbar. To display the Font Toolbar, illustrated below, choose Toolbars from the View menu and then choose Font Toolbar.

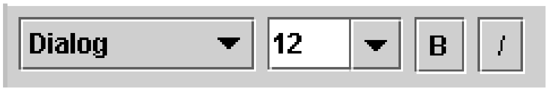


Figure 8.11 *The Font Toolbar*

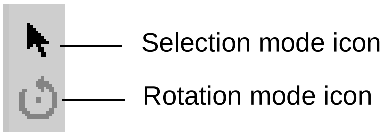
As with colors, this change applies both to the selected objects of the active buffer and to the default font for newly created objects.

Changing the Ratio of Rectangle Objects

You can change the default ratio of the dimensions for the rectangular objects you create. This is done on the Edit page of the Options window. To open this window, choose Options from the Tools menu, then set the ratio in the “Aspect ratio of rectangular interactors” field. As an example, typing `3` in the field defines rectangles that are three times longer than they are high.

Editing Modes

The editing modes are represented by their icons on the left toolbar:



Selection Mode

The Selection mode lets you select, move, resize, and perform common editing operations. To activate the selection mode, click its icon in the left toolbar.

The different functionalities of the selection mode and how to use them are listed below:

- **Select an object:** Click on the object.
- **Select/deselect multiple objects:** Hold down the Shift key and click on the objects.
- **Select multiple objects in one operation:** Drag a selection box around the objects you want to select.
- **Deselect all selected objects:** Click in the background of the view.
- **Mark a particular object:** Hold down the Shift and Ctrl keys and click on that object. The selection handles of this object will then appear in red. When it is deselected, it will be automatically unmarked. Marking an object can be required by some Composer editing facilities.
- **Move an object:** Drag the object. You can change the way of moving objects to opaque move instead of the default ghost move. This is done on the Edit page of the Options window. To access this window, choose the Options command from the Tools menu.

  For more information on the ghost move, see the `drawGhost` method examples in the section *Example: Extending the IlvObjectInteractor Class* on page 69.
- **Edit the shape of an object:** Drag the selection handles, for objects of rectangular shape and ellipses. For objects such as lines, polylines, and splines, drag the control points.
- **Pan the view:** By pressing the Spacebar and dragging the mouse you can pan the view.
- **Magnify a part of the view:** By pressing the Ctrl and Alt keys and dragging the mouse you can magnify a part of the view.

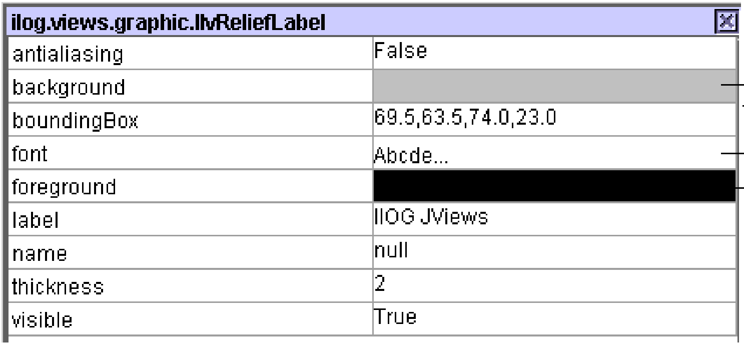
> ***Note:*** *The Select All command in the Edit menu allows you to select all objects in a buffer at once.*

Rotation Mode

The second editing mode is the Rotation mode. It lets you rotate a selected object by dragging one of its selection handles with the mouse pointer.

Inspecting and Editing Objects in the Properties Sheet

The Property Sheet allows you to inspect as well as edit the properties of objects. To display this window, choose the Property Sheet command from the View menu. Once the Property Sheet is displayed, click an object to display its properties. To change a property, enter the new value in the value field to the right of the property and press the Enter key.



ilog.views.graphic.IlvReliefLabel

antialiasing	False
background	▨
boundingBox	69.5,63.5,74.0,23.0
font	Abcde...
foreground	■
label	ILOG JViews
name	null
thickness	2
visible	True

Properties that require a special editor.

Figure 8.12 Property Sheet for an Object of the Class IlvReliefLabel

Editing Properties that Require a Special Editor

The properties listed below require a special editor:

- **foreground:** To modify the color of text, borders, shadows and so on of an object.
- **font:** To modify the font (can also be carried out directly from the Font Toolbar).
- **background:** To modify the fill-in or pattern color of the object, as well as the graphic portion of a text object.
- **paint:** To modify the fill-in or pattern color of General Path objects.
- **labels:** To add or remove a label from a scale object.
- **stroke:** To modify the border of General Path objects.

To change these properties, follow the procedure below (note that a single click is specified):

1. Click once in the value field to the right of the property you want to change.
2. Click once on the editor button that appears: ...
3. Change the property in the editor.
4. Click Apply.

Editing Multiple Objects in the Property Sheet

You can change the properties of several objects selected at once. The Property Sheet displays the common properties of the selected objects. In the case where the properties do not have the same value, the values are grayed. They can still be changed, however; simply click in the fields and enter the new values.

Editing Properties of Wrapped Objects

The selected `IlvGraphic` may be a wrapper object (an `IlvTransformedGraphic`, for instance) that has been created using one of the edit commands described in *Editing Objects* on page 150. In this case if you do not want to edit the properties of the wrapper object but rather of the wrapped one, you can display the pop-up menu on the property sheet and choose the “deep properties” item, which will allow you to edit the properties of the wrapped object.

Editing Objects

Aside from the Property Sheet window, Composer offers the following features to edit objects that have already been created.

- *Grouping and Ungrouping Selected Objects*
- *Cutting, Copying, and Pasting Selected Objects*
- *Hiding and Showing a Selection of Objects*
- *Changing the Layer of an Object*
- *Aligning, Distributing, or Flipping Selected Objects*
- *Changing Objects to an IlvGraphic Wrapper Instance*
- *Undoing/Redoing*
- *Editing General Path Objects*

Grouping and Ungrouping Selected Objects

The editor allows you to group objects together. To group a set of objects, select all the objects and choose Group As Set or Group As Manager from the Edit menu. A new object is created that contains all the grouped objects.

When grouping objects, if you choose from the Edit menu:

- Group As Set, a new `IlvGraphicSet` object is created. Once objects are grouped inside an `IlvGraphicSet` object, you can no longer modify the objects within. You can only move and resize the group.
- Group As Manager, a new `IlvGrapher` object is created. If the objects are grouped inside an `IlvGrapher`, it is still possible to modify objects within.

To ungroup objects, select the group they belong to and choose UnGroup from the Edit menu.

These functionalities are also available in the contextual pop-up menu of the buffer.

Cutting, Copying, and Pasting Selected Objects

The selected objects can be cut and copied using the Cut and Copy commands in the Edit menu or the contextual pop-up menu in the buffer. Once cut or copied, the selected objects can be pasted in the selected buffer, which can be the one from which the copy was made or another one.

If the copied objects were in a nested manager of the buffer, they are pasted as follows:

- By default the objects are pasted in the same nested manager of the buffer.
- However, if another nested manager is marked (see *Selection Mode* on page 148), the copied objects will be pasted in the marked nested manager.
- If nothing is selected, the objects will be pasted in the top-level manager of the buffer.

Hiding and Showing a Selection of Objects

If you want to hide a certain selection of objects in different layers, select the objects and set their Visible property to `False` in the Property Sheet. To reverse the procedure, set the Visible property back to `True`. You may also use the Show Hidden Objects command in the Edit menu if you have no objects that you want to remain hidden.

> ***Note:*** *If you want to hide or show all objects of a given layer, use the Visible toggle button in the Layers Editor window. See Specifying Layers for a Buffer on page 145.*

Changing the Layer of an Object

Layer0 is the insertion layer by default. If you change the insertion layer in the Layers window, as explained in the section *Specifying Layers for a Buffer* on page 145, the objects created afterwards are placed in the newly specified layer. If you want to change the layer of an object that has already been created, you can specify a layer or move the object with the Arrange operation.

To precisely set the layer:

1. Select the object. If the object is in a nested manager, you have to open a view on the nested manager before being able to edit the layer of the object (see *Creating Buffers* on page 128).
2. Open the Layers Editor window by choosing the Windows command from the View menu and then selecting the Layers Editor item, if it is not already opened.
3. Change the layer in the Selected Objects Layer combo box.

Another solution is to use an Arrange command, which allows you to move an object backward or forward in the layer hierarchy. The Arrange commands are available under the Arrange submenu of the Edit menu or the contextual pop-up menu of the buffer.

Aligning, Distributing, or Flipping Selected Objects

To align, distribute, or flip objects, click the corresponding icons on the Alignment toolbar. To access this toolbar, select Toolbars in the View menu and then check the Alignment Toolbar check box.

These commands and their corresponding icons are listed in the table below along with a description.

Table 8.4 *Alignment Toolbar Icons*

	Align Left: aligns the selected objects on the left side of their global bounding box
	Align Horizontal Center: shifts the objects horizontally so that they are aligned on the vertical median of their global bounding box
	Align Right: aligns the selected objects on the right side of their global bounding box
	Align Top: aligns the selected objects at the top of their global bounding box
	Align Vertical Center: shifts the objects vertically so that they are aligned on the horizontal median of their global bounding box
	Align Bottom: aligns the selected objects at the bottom of their global bounding box
	Distribute Horizontal: shifts the objects horizontally, positioning them at equal distance from each other in their global bounding box
	Distribute Vertical: shifts the objects vertically, positioning them at equal distance from each other in their global bounding box

Table 8.4 Alignment Toolbar Icons (Continued)

	Flip Vertical: flips an object vertically
	Flip Horizontal: flips an object horizontally
	Rotate Right: rotates an object to the right by ninety degrees
	Rotate Left: rotates an object to the left by ninety degrees

These commands are also available in the Edit menu and in the contextual pop-up menu of the view.

Changing Objects to an IlvGraphic Wrapper Instance

If you need to change a selected object to a fixed-size graphic or a transformed graphic wrapper instance, you can use the Convert to Graphic Handle sub-menu from the Edit menu. The items of the submenu with their descriptions follow:

- **Fixed Size:** Transforms selected objects into an `IlvFixedSizeGraphic` object.
- **Transformed:** Transforms selected objects into an `IlvTransformedGraphic` object.

To reverse these operations, choose the Revert from Graphic Handle command from the Edit menu.

Undoing/Redoing

Most of the creation and editing operations on graphic objects can be undone or redone using the corresponding command from the Edit menu or their corresponding icons on the toolbar. The undo/redo buffer limit can be changed on the Edit page of the Options window. To access this window, choose the Options command from the Tools menu.

Editing General Path Objects

Additional editing options are available for General Path objects in the Editing Shapes submenu of the Edit menu. You can perform the following operations:

- **Unions**: by selecting a set of General Path objects (one by clicking it and the other by pressing the Ctrl key when clicking it) and using the Shapes Union command of the Edit > Editing General Path menu, a General Path representing the union of their shapes will replace the set of General Path objects.

- **Intersections**: by selecting a set of General Path objects with intersections and using the Shapes Intersection command, a General Path representing the intersection of their shapes will replace the set of General Path objects.
- **Subtractions**: by selecting two General Path objects and using the Shapes Subtraction command, a General Path is created representing the subtraction of the shape of the marked General Path (see *Selection Mode* on page 148) from the shape of the regularly selected one.
- **Flattening**: by selecting a set of General Path objects and using the Flat Shapes command, the shapes of the selected General Path will be flattened.

Controlling Buffers

The following sections present the commands for controlling the buffers.

Zooming In and Out

- To zoom in on the contents of the main panel:

  Click the Zoom In icon (🔍+) on the left toolbar or use the keyboard accelerators SHIFT+Z.

- To zoom out:

  Click the Zoom Out icon (🔍−) on the left toolbar or use the keyboard accelerators SHIFT+U.

The Scale 1:1 icon (1:1) brings you back to the original state of the buffer after having carried out transformations on it, such as zooming.

Changing Buffer Options

General buffer options can be changed on the Edit page of the Options window. To access this window, choose the Options command from the Tools menu. For example, you can put buffers in anti-aliasing mode or force them to keep the aspect ratio when zooming the buffer in or out.

Printing and Saving Files

You can print graphic objects using the Print command from the file menu or with the printing icon on the toolbar (🖨). If you want to specify how many number of pages you want to print, you can use the Page Setup command from the same menu. Finally, you have the ability to specify only a portion of the buffer to print it using the File > Print Area > Set Print Area command by dragging a rectangle representing that area on the active buffer.

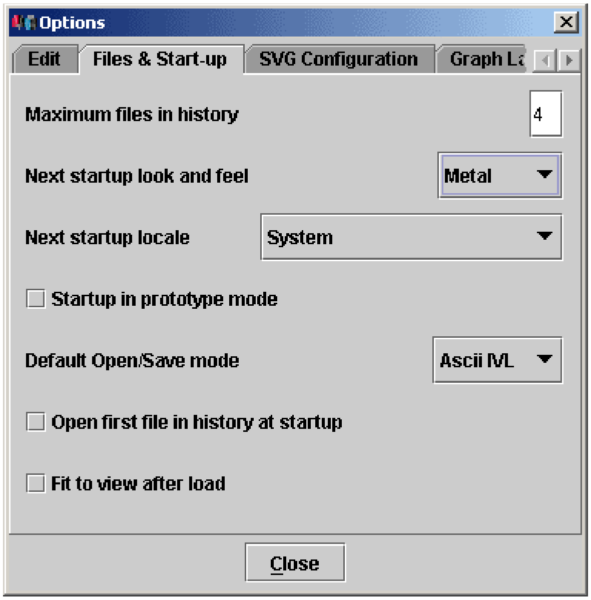
In addition to the usual printing mechanism, you can also dump the contents of the editing buffer to a PNG or JPEG file by using the Export as Bitmap command from the File menu.

To save files, use the Save As command from the File menu or the Save (💾) icon on the toolbar. Composer saves files in the `.ivl` ASCII format by default. However, you can also select the `.ivl` Binary or SVG mode in the Files of type combo box.

See also the section *Saving Parameter Settings* on page 136.

Setting Files and Startup Options

Before quitting Composer, you can specify certain options which will be taken into account the next time you run Composer. This is done on the Files & Start-up page of the Options window. To access this window, choose Options from the Tools menu.



Maximum files in history: 4

Next startup look and feel: Metal

Next startup locale: System

☐ Startup in prototype mode

Default Open/Save mode: Ascii IVL

☐ Open first file in history at startup

☐ Fit to view after load

Figure 8.13 *Files & Start-up Page*

The options which may be specified are listed below:

- **Maximum files in history:** Sets the number of history files (those last saved in Composer) that are displayed for quick access in the File menu.

- **Next startup look and feel:** Sets the look of the Composer interface.
- **Next startup locale:** Sets the language (if available) that will be used to display menus and messages.
- **Startup in prototype mode:** Starts Composer with the Prototype module.
- **Default Open/Save mode:** Changes the default mode for loading and saving files.The option is taken into account for the buffers that are created after modification.
- **Open first file in history at startup:** Opens the last file used.
- **Fit to view after load:** Fits the buffer content to its size after loading an IVL file.

Exiting Composer

To exit Composer, choose Exit from the File menu.

When exiting Composer, many basic options set during your work (the display of toolbars for example) are saved in an XML file and are activated when opening for the next work session.

9

Graphics Framework JavaBeans

The main classes of ILOG Views Graphics Framework fully comply with the JavaBeans^{TM} standard. This allows you to create an ILOG JViews application from the visual programming environment of your favorite Integrated Development Environment (IDE).

This chapter features an example that shows you how to use the ILOG JViews Beans when creating an applet within an IDE. At the same time, you will see the main functionality of the Beans. The following topics are covered:

- *Installing ILOG JViews Beans into an IDE*
- *Graphics Framework Classes Available as JavaBeans*
- *Creating a Simple Applet Using ILOG JViews Beans*

Installing ILOG JViews Beans into an IDE

To be able to use the ILOG JViews Graphics Framework Beans, you must first install the Beans into your IDE.

The Beans are located in the `jviewsall.jar` file of the `<installdir>/classes` directory.

In addition to the ILOG JViews classes, this `.jar` file contains classes that are required when the Beans are being edited in the IDE, such as property editors or customizers. However, these classes are not required to run the applications.

Instructions are given below to install the Beans into Inprise JBuilder 7^{TM}. For other IDEs, refer to their documentation. In most cases, the IDE simply allows you to read a `.jar` file and finds the JavaBeans located in this `.jar` file automatically.

Installing Beans into Inprise JBuilder 5

For this installation:

1. Choose Configure Libraries from the Tools menu.

   The Libraries dialog box appears.

2. Choose New to define a new library.

3. In the New Library Wizard, define a new Library named `JViews` with the path pointing to the `jviewsall.jar` JAR file of ILOG JViews.

4. Click OK to close the dialogs.

5. Choose Configure Palette from the Tools menu.

   The Palette Properties dialog box appears.

6. Select the Pages tab.

7. Click Add to create a new page named `JViews`.

8. Select the Add components tab.

9. Click Select library and choose the JViews library.

   The Component filtering must be on 'JavaBeans in jar manifest only.'

10. Click Add from Selected library.

    A dialog box named Results shows you the Beans that have been added.

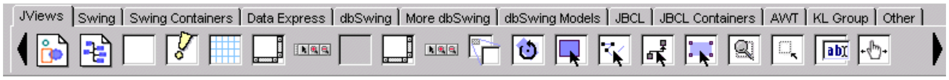


Figure 9.1 *The ILOG JViews Beans in JBuilder 5*

> ***Note:*** *ILOG JViews requires JDK 1.3 to run. Some IDEs may refuse to import these ILOG JViews Beans because they are running earlierJDK versions. In this case, you may need a newer version of your IDE.*

When you import the JavaBeans into your IDE, you will see that the `jviewsall.jar` file contains Beans related to the Graph Layout, Maps, or Gantt Chart packages of ILOG JViews. These Beans are not covered in this manual. To learn more about these Beans, read the section about JavaBeans in the *User's Manual* of the corresponding module.

Graphics Framework Classes Available as JavaBeans

The ILOG JViews Graphics Framework provides the following groups of Beans:

- *ILOG JViews Main Data Structure Beans*
- *ILOG JViews Main GUI Components*
- *Predefined Interactors*
- *GUI Convenience Components*

These Beans are classes of the ILOG JViews library. The details of these classes are explained throughout this manual as well as in the *ILOG JViews Graphics Framework Reference Manual*. These Beans are listed below along with their icons that are displayed on the toolbar.

> ***Note:*** *Either the small icon or the large icon is displayed depending on the IDE you use.*

ILOG JViews Main Data Structure Beans

The **`IlvManager`** Bean, the data structure that stores the graphic objects. In this Bean, you can specify the initial `.ivl` file to be loaded.

The **`IlvGrapher`** Bean, which organizes the graphic objects into nodes and links of a network.

ILOG JViews Main GUI Components

All the ILOG JViews GUI components needed to create an AWT or JFC/Swing applet or application are available as JavaBeans:

The **`IlvManagerView`** Bean, the visual Bean that displays the content of a manager Bean.

The **`IlvJScrollManagerView`** Bean, a Swing-based Bean that adds the scrolling functionality to `IlvManagerView` objects.

The **`IlvScrollManagerView`** Bean, the AWT version of the `IlvJScrollManagerView`.

The **`IlvManagerViewPanel`** Bean, an AWT component designed to contain an `IlvManagerView` Bean and to manage the double-buffering mechanism of the manager view. This Bean is necessary only to create AWT applets or applications using double-buffering.

The **`IlvGrid`** Bean, the magnetic grid that can be installed on any `IlvManagerView` Bean.

Predefined Interactors

The predefined interactors provided as JavaBeans are given below:

The **`IlvManagerViewInteractor`** Bean, an interactor Bean that has no predefined interaction. You can create your own interaction by binding the different input events (mouse, keyboard) sent by this Bean.

The **`IlvZoomViewInteractor`** Bean, an interactor that allows the user to select a rectangle of a manager view to be zoomed in.

The **`IlvSelectInteractor`** Bean, an interactor that allows the user to select and edit the graphic objects of a manager.

The **`IlvPanInteractor`** Bean, an interactor that allows the user to pan the view of a manager.

The **`IlvRotateInteractor`** Bean, an interactor that allows the user to rotate objects in a manager.

The **`IlvManagerMagViewInteractor`** Bean, an interactor that controls the panning and zooming of a target view by manipulating a control rectangle on the view.

The **`IlvDragRectangleInteractor`** Bean, an interactor that allows the user to drag a rectangle on a view. You can perform any type of action when the rectangle is dragged by binding the `RectangleDragged` event.

The **`IlvMakeRectangleInteractor`** Bean, an interactor that allows the user to create any type of rectangular object in a manager.

The **`IlvMakePolyPointsInteractor`** Bean, an interactor that allows the user to create any type of graphic object defined by a set of points, such as a polyline or spline.

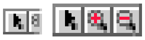
The **`IlvEditLabelInteractor`** Bean, an interactor that allows the user to create and edit a graphic object that contains a label.

The **`IlvMakeLinkInteractor`** Bean, an interactor that allows the user to create a link in a grapher.

The **`IlvMagnifyInteractor`** Bean, an interactor that allows the user to move a lens over the view of a manager to magnify the objects under it.

GUI Convenience Components

The **`IlvJManagerViewControlBar`** Bean, a Swing toolbar that allows the user to perform selection, zoom, and pan operations on an `IlvManagerView` Bean.



The **`IlvManagerViewControlBar`** Bean, an AWT version of the `IlvJManagerViewControlBar` Bean.

Creating a Simple Applet Using ILOG JViews Beans

In this section you will create a simple ILOG JViews applet using ILOG JViews Graphics Framework Beans. No coding will be necessary. The applet you create is a simple JFC/ Swing applet that displays a butterfly with a toolbar allowing you to zoom and pan the content of the view. This section does not explain the concepts that underlie JavaBeans. For this information, refer to the Web site: `http://java.sun.com/products/javabeans`. Furthermore, we assume that you are familiar with the manipulation of JavaBeans inside your IDE.

> ***Note:*** *The Swing Beans that you will use have the letter "J" in the prefix of the Bean name. You could also create the same type of application using only AWT controls. To do so, you would simply use the* `IlvScrollManagerView` *Bean that is an AWT control instead of the* `IlvJScrollManagerView` *Bean.*

The following example is carried out using a typical IDE procedure. It contains the following steps:

- *Creating the Manager View*
- *Setting the Properties of the Manager View*
- *Creating a Manager and Displaying its Content in a View*
- *Loading an .ivl File into the Manager*
- *Adding a Control toolbar Bean*
- *Configuring the toolbar*
- *Testing the Result*

Creating the Manager View

To create the manager view:

1. Create a new project as a JFC/Swing applet or application.

2. Display the ILOG JViews Beans on the toolbar by selecting that package.

3. From the toolbar, click the `IlvJScrollManagerView` Bean icon and drag it inside the form designer of your IDE.

> ***Warning:*** *There are two of these icons on the toolbar. Make sure you are using* `Ilv`**`J`**`ScrollManagerView` *and not* `IlvScrollManagerView`.

4. Drag the handles of your `IlvJScrollManagerView` Bean until it appears as follows:

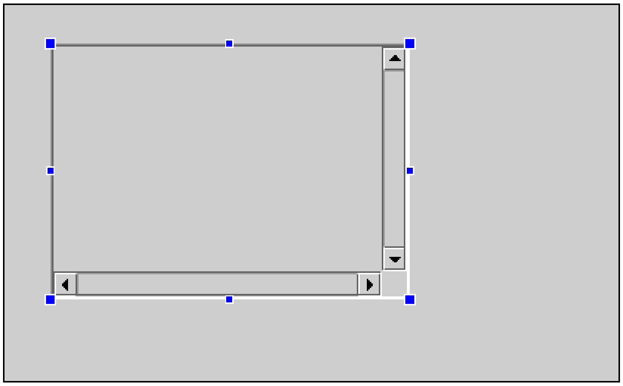


Figure 9.2 `IlvJScrollManagerView` *Object Selected in the Form Designer*

5. Click the `IlvManagerView` Bean icon on the toolbar and drag it inside of the `IlvJScrollManagerView` Bean.

   The result is fairly similar to what you obtained after the previous step, except that you can now select the manager view. See the figure below.

> ***Note:*** *If you were to compile and run the project at this point, you would see that the* `IlvJScrollManagerView` *allows you to scroll through the content of the* `IlvManagerView` *Bean.*

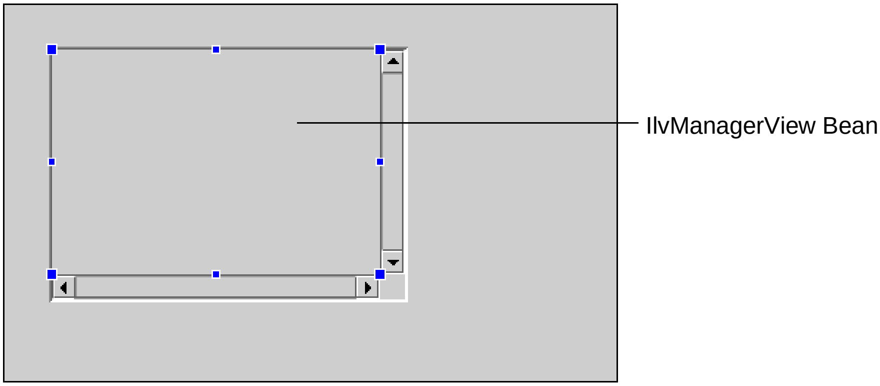
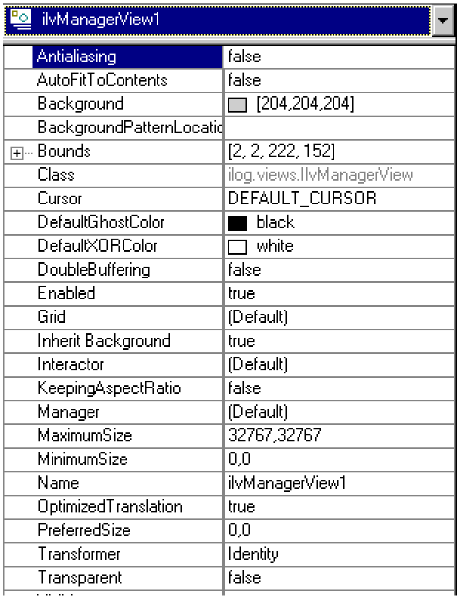


Figure 9.3 *`IlvJScrollManagerView` Object with a Selected `IlvManagerView` Object Inside*

Setting the Properties of the Manager View

You are now going to change a manager view property of the Bean, which is done in the property sheet illustrated below. This property sheet is active because the `IlvManagerView` object is presently selected in the form designer.



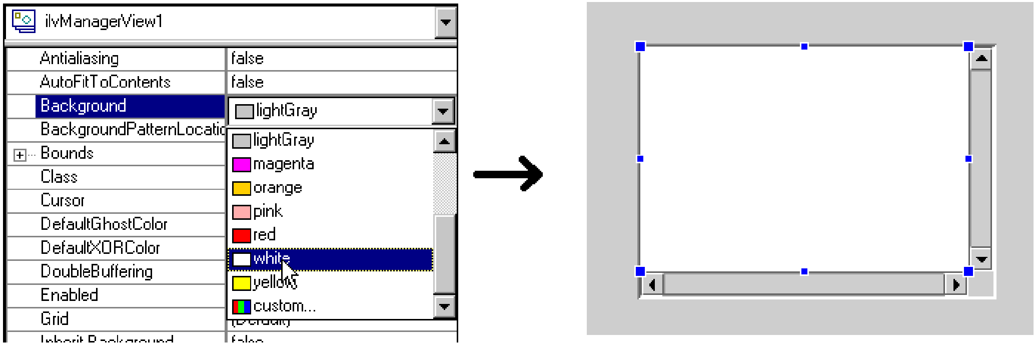
ilvManagerView1

Antialiasing	false
AutoFitToContents	false
Background	▨ [204,204,204]
BackgroundPatternLocatic	
⊞ Bounds	[2, 2, 222, 152]
Class	ilog.views.IlvManagerView
Cursor	DEFAULT_CURSOR
DefaultGhostColor	■ black
DefaultXORColor	□ white
DoubleBuffering	false
Enabled	true
Grid	(Default)
Inherit Background	true
Interactor	(Default)
KeepingAspectRatio	false
Manager	(Default)
MaximumSize	32767,32767
MinimumSize	0,0
Name	ilvManagerView1
OptimizedTranslation	true
PreferredSize	0,0
Transformer	Identity
Transparent	false

Figure 9.4 *Property Sheet for `IlvManagerView` Object*

The property you want to change is the background property:

1. Click the value field of the `Background` property and change the background of the view to `white`:



ilvManagerView1

Antialiasing	false
AutoFitToContents	false
Background	lightGray
BackgroundPatternLocatio	
Bounds	
Class	
Cursor	
DefaultGhostColor	
DefaultXORColor	
DoubleBuffering	
Enabled	
Grid	(Default)
Inherit Background	false

Figure 9.5 Setting the Background Property of a View

2. Change the `KeepingAspectRatio` property to `true`.

   This will ensure that the zoom level remains the same along the x and y axis.

Creating a Manager and Displaying its Content in a View

You can now create an `IlvManager` Bean. The `IlvManager` Bean provides the data structure that contains the graphic objects you want to display.

To create the `IlvManager` Bean:

1. Click the `IlvManager` Bean icon on the toolbar.
2. Drag it into the form designer.

   The class `IlvManager` is not a graphical Bean, so it is not managed the same way by the different IDEs. The image below shows the manager as a small object inside the form designer.

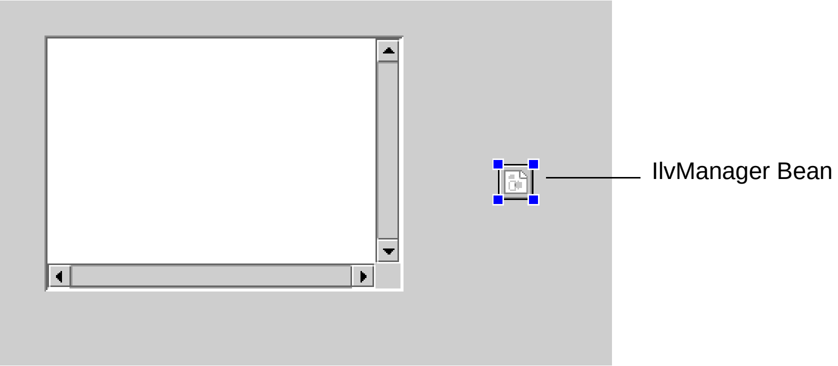
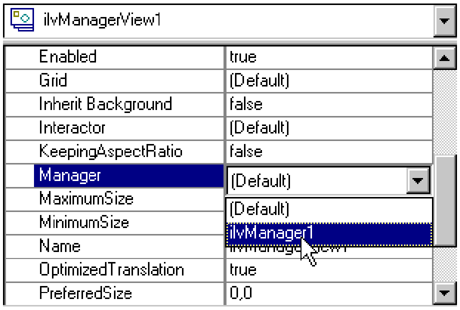


Figure 9.6 *The* `IlvManager` *Bean in the Form Designer*

You must now associate the view with the manager. This is done by setting the manager property of the `IlvManagerView` Bean to our new manager Bean.

3. Select the `IlvManagerView` object so that its property sheet is active.

4. Set the value of its `Manager` property to `ilvManager1` as seen below:



ilvManagerView1

Enabled	true
Grid	(Default)
Inherit Background	false
Interactor	(Default)
KeepingAspectRatio	false
Manager	(Default) (Default) ilvManager1 ilvManagerView1
MaximumSize	
MinimumSize	
Name	
OptimizedTranslation	true
PreferredSize	0,0

Figure 9.7 *Setting the Manager Property of a View*

The `IlvManagerView` will now display the content of the `IlvManager` Bean. You can create several `IlvManagerView` objects and associate them with the same `IlvManager` Bean. This allows you to have several views of the same data.

Loading an .ivl File into the Manager

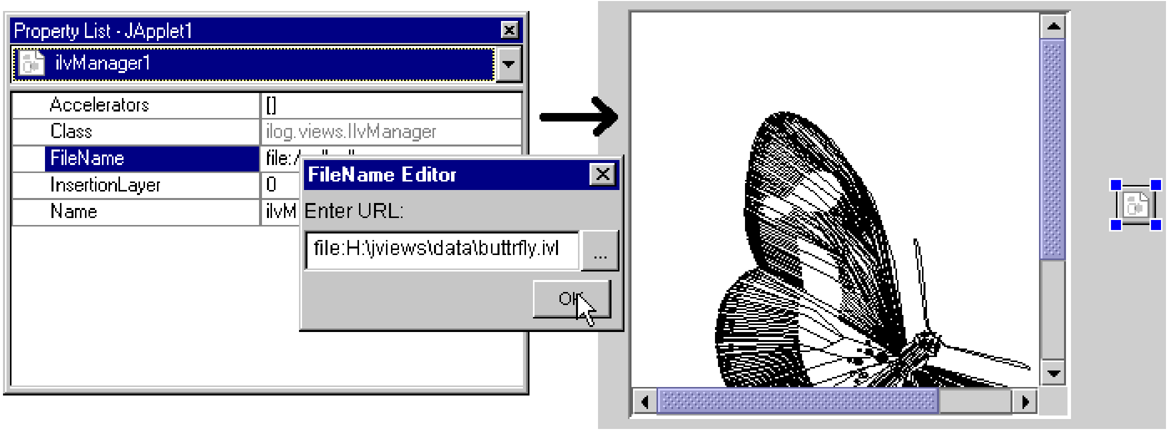
You can now load an `.ivl` file into the `IlvManager` Bean.

To do so:

1. Select `IlvManager1` so that its property sheet is active.

2. Click in the value field of the `FileName` property and then click the ...button that appears.
3. Click the ...button in the FileName Editor window that appears.
4. Browse to the `buttrfly.ivl` file located in the `<installdir>/data` directory and open it.
5. Click OK in the FileName Editor dialog box.

   The file is automatically displayed in the `IlvManagerView` Bean.



Accelerators	[]
Class	ilog.views.IlvManager
FileName	file:/
InsertionLayer	0
Name	ilvM

Enter URL: file:H:\jviews\data\buttrfly.ivl

Figure 9.8 Loading a File into the Manager

Adding a Control toolbar Bean

You will now add a toolbar that allows the user to control the zoom level of the view and to pan the view.

To do so:

1. Click the `IlvJManagerViewControlBar` icon on the ILOG JViews Beans toolbar.
2. Drag it into the form designer.

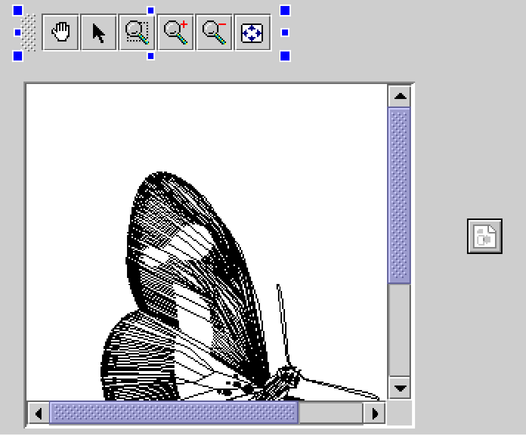
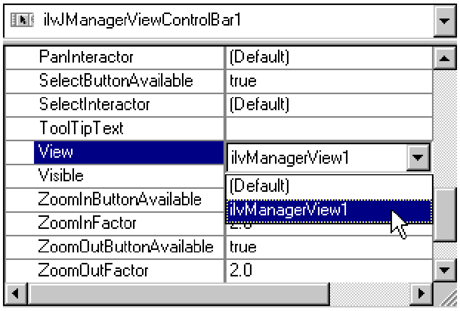


Figure 9.9 *The Control toolbar in the Form Designer*

You must now associate the toolbar with the view by setting the `View` property of the toolbar.

3. Verify that the `IlvJManagerViewControlBar` object is selected so that its property sheet is active.
4. Select `ilvManagerView1` in the value field of the `View` property as seen below:



ilvJManagerViewControlBar1

PanInteractor	(Default)
SelectButtonAvailable	true
SelectInteractor	(Default)
ToolTipText	
View	ilvManagerView1
Visible	
ZoomInButtonAvailable	
ZoomInFactor	2.0
ZoomOutButtonAvailable	true
ZoomOutFactor	2.0

Figure 9.10 *Associating the toolbar with the View*

Configuring the toolbar

You may configure the toolbar in different ways. You can:

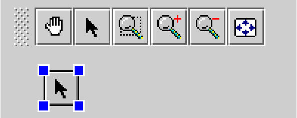
- Hide some of the predefined button icons of the toolbar by setting the corresponding properties: `PanButtonAvailable`, `SelectButtonAvailable`, and so on.
- Also add your own button icons to the toolbar, as you can with any Swing toolbar.

- Modify the default interactors that are used in the toolbar.

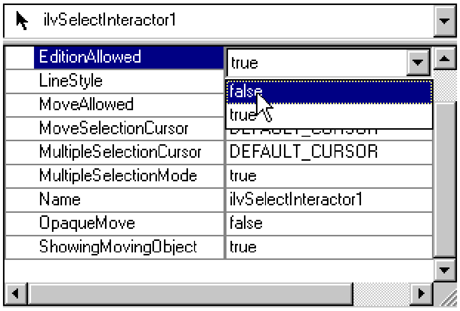
For example, the toolbar has a `selectInteractor` property that allows you to change the selection interactor used in the toolbar when clicking on the Select button icon. You can modify the properties of the selection interactor Bean to define the type of selection you need. For example, you may want to disable the editing capability.

To do this:

1. Click an `IlvSelectInteractor` Bean ↖ on the toolbar and drag it into the form designer.



2. Set its `EditionAllowed` property to `false` as seen below.



ilvSelectInteractor1

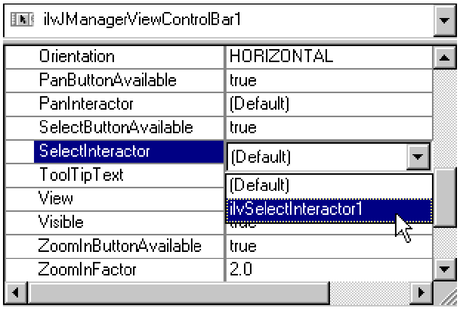
EditionAllowed	true
LineStyle	
MoveAllowed	
MoveSelectionCursor	DEFAULT_CURSOR
MultipleSelectionCursor	DEFAULT_CURSOR
MultipleSelectionMode	true
Name	ilvSelectInteractor1
OpaqueMove	false
ShowingMovingObject	true

Figure 9.11 *Customizing the Selection Interactor.*

You are now going to replace the default selection interactor used in the toolbar by setting the `SelectInteractor` property of the `ilvJManagerViewControlBar1`.

3. Select the `ilvJManagerViewControlBar1` object so that its property sheet is active.

4. Change the value of the `SelectInteractor` property to `ilvSelectInteractor1` as seen below:



ilvJManagerViewControlBar1

Orientation	HORIZONTAL
PanButtonAvailable	true
PanInteractor	(Default)
SelectButtonAvailable	true
SelectInteractor	(Default)
ToolTipText	
View	
Visible	true
ZoomInButtonAvailable	true
ZoomInFactor	2.0

Figure 9.12 Replacing the Default Selection Interactor

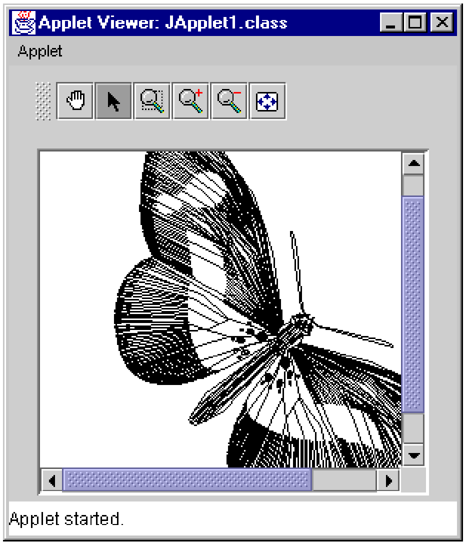
Result

Compile the project. You have created a Java application without writing a single line of code.

> ***Note:*** *In this example, you have added interaction to the view by means of the control toolbar. You could also directly set an interactor Bean such as the* `IlvSelectInteractor` *on the manager view by using the interactor property of the* `IlvManagerView`*.*

Testing the Result

Execute the applet. The resulting application should be as follows:



Applet started.

Figure 9.13 Final Application

Use the scroll bars and the following toolbar icons to manipulate the image displayed in the manager view:

- The Pan icon to pan the content of a view.
- The Select arrow icon to select objects in the view.
- The Interactive zoom icon to drag a rectangle over an area that you want to zoom.
- The Zoom-in and the zoom-out icons .
- The Fit to view icon to ensure that the content of the manager is fully displayed.

This concludes the example. For information on how to save your project and to know what type of files are generated when saving, refer to the documentation of your IDE.

CHAPTER

10

Thin-Client Support in ILOG JViews Graphics Framework

The ILOG JViews class library can be used on the client side where you develop Java applets or applications. It can also be used on the server side. Some “Web browser” applications require that the client stay very light, with most of the functionality residing in the server. The “thin-client” support in ILOG JViews Graphics Framework allows you to create such types of applications easily. You can use the power of the ILOG JViews class library to build complex two-dimensional representations on the Web server and use the ILOG JViews Graphics Framework thin-client support on your Web browser to display and interact with those images created by the server.

In this chapter we will see how to use the ILOG JViews packages and classes on both the server side and the client side. The topics are:

- *ILOG JViews Thin-Client Web Architecture*
- *Getting Started With the ILOG JViews Thin Client: an Example*
- *Developing the Server Side*
- *Developing the Client Side*
- *Adding Client/Server Interactions*
- *Generating a Client-Side Image Map*
- *The IlvManagerServlet Class*

ILOG JViews Thin-Client Web Architecture

The ILOG JViews thin-client support is based on the *Java servlet* technology. Servlets are Java programs that run on a Web server. They act as a middle layer between HTTP requests coming from a Web browser or other HTTP clients such as applets or applications and the application or databases on the Web server. The job of the servlet is to read and interpret HTTP requests coming from an HTTP client program and to generate a resulting document that in most cases is an HTML page.

For more information about servlet technology, you can visit the JavaSoft site `http://java.sun.com/products/servlet`.You will also find their information about the Web servers supporting Java servlets.

For the predefined types of ILOG JViews clients, the content created by the servlet is primarily a JPEG image. On the client side, user interactions with the image are managed by code in one of two formats: Java applets or Dynamic HTML scripts.

Creating a Web application with ILOG JViews consists of using the ILOG JViews library on the server side to create complex two-dimensional displays based on application data that resides on the server. A servlet will answer HTTP requests from a client and deliver images to this client, as illustrated in Figure 10.1.

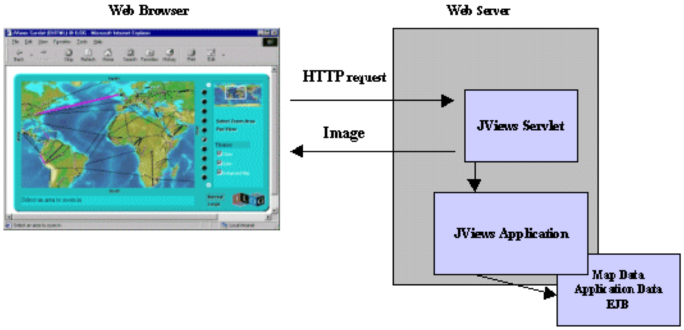


Figure 10.1 *Client-Server Display Interaction*

The ILOG JViews Graphics Framework thin-client support contains the following:

- An abstract servlet class that can generate JPEG images from an ILOG JViews display.
- A set of Dynamic HTML scripts written in JavaScript that will be used on the client side to display and interact with the image created on the server side.
- A set of JavaBeans that will be used to create a thin Java applet that displays and interact with the image created on the server side.

Getting Started With the ILOG JViews Thin Client: an Example

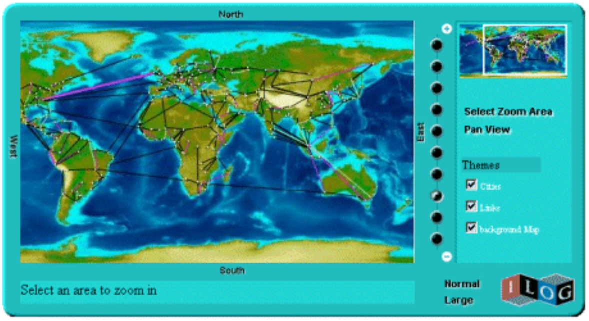
Creating an ILOG JViews thin-client application consists of two steps: developing the server side and developing the client side. An example is provided with this release to illustrate these steps. **XML Grapher** shows how to build the server side and also how to create a Dynamic HTML client and a thin Java client.

The XML Grapher Example

The XML Grapher example can be found in the following directory:

`<installdir>/demos/servlet/xmlgrapher`

This example allows you to show a network of interconnected cities on top of the map in a thin-client context.



☑ Cities
☑ Links
☑ background Map

Select an area to zoom in

Figure 10.2 *The XML Grapher Example*

The XML Grapher example is composed of the following pieces:

- An ILOG JViews component that can read an XML file describing a set of interconnected cities and display them on top of a map as shown in the picture above.

  This component is located in the files:

  `<installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher/XmlGrapher.java`

  `<installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher/GrapherNode.java`

- Some example XML files for the component, located in:

  `<installdir>/demos/servlet/xmlgrapher/webpages/data`

- A servlet that can produce JPEG images from the component described above.

  The servlet is located in:

```
<installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher
  /servlet/XmlGrapherServlet.java
```

- A Dynamic HTML client composed of:
  - The HTML starting page:

    `<installdir>/demos/servlet/xmlgrapher/webpages/dhtml/index.html`
  - The set of JavaScript Dynamic HTML components, located in:

    `<installdir>/classes/thinclient/javascript`
  - Some images required for the example, located in:

    `<installdir>/demos/servlet/xmlgrapher/webpages/dhtml/images`
- A Thin Java client composed of:
  - The HTML starting page:

    ```
    <installdir>/demos/servlet/xmlgrapher/webpages/applet/
    index.html
    ```
  - The JAR file of the applet and the JAR file of ILOG JViews thin-client support:

    ```
    <installdir>/demos/servlet/xmlgrapher/webpages/applet
      /xmlgrapher.jar
    ```

    `<installdir>/classes/jviewsthin.jar`
  - Some images required for the applet in:

    `<installdir>/demos/servlet/xmlgrapher/webpages/applet/images`
  - The source of the applet, located in:

    ```
    <installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher
      /applet/XmlGrapherApplet.java
    ```

Installing and Running the Example

To be able to run, this example requires a Web server and a Web browser that supports Dynamic HTML (for the DHTML client) and Java 1.1 (for the thin Java applet client).

Only the following releases of Web browsers will support DHTML: Netscape Communicator 4.x (on Windows and UNIX platforms) and Internet Explorer 5.0 and higher (on Windows platforms).

The example contains a WAR (Web ARchive) file that allows you to easily install the example on any server that supports the Servlet API 2.1 or later. For your convenience, the WAR file has already been installed for you into the TOMCAT Web server that is supplied with the ILOG JViews installation. TOMCAT is the official reference implementation of the Servlet and JSP specifications. If you are already using an up-to-date Web or application server, there is a good chance that it already has everything you need. You can check the latest list of servers that support servlets at: `http://java.sun.com/products/servlet/industry.html`.

Here are the steps for running the example on the TOMCAT Web server supplied with the ILOG JViews installation:

1. Set the JAVA_HOME environment variable to point to your Java Development Kit installation.
2. Go to the TOMCAT `bin` directory located in `<installdir>/tools/tomcat/bin`, where `<installdir>` is the directory in which ILOG JViews is installed.
3. Depending on your system, run the `startup.bat` or `startup.sh` script to run the TOMCAT server.
4. To see the example, launch a Web browser and open the page:

   `http://localhost:8080/xmlgrapher/index.html`

   This page gives you access to two different clients: a Dynamic HTML client and a thin Java client.

> ***Note:*** *You must use* `localhost` *instead of the name of your machine. Otherwise, the sample applet may not be able to connect to the servlet.*

Running the Examples Without an X Server

The ILOG JViews servlets can run with the headless support that is built into JDK 1.4. For more information on this feature, please refer to the JDK Release Notes.

If you are using an earlier version of the JDK (3.1.x), you can run a virtual X Server such as `xvfb` (X Virtual Frame Buffer). This solution is described on the Sun Web site.

Developing the Server Side

The server side of an ILOG JViews thin-client application is composed of two main parts: the ILOG JViews application itself, which can be any type of complex two-dimensional display built on top of the ILOG JViews API, and a Servlet that produces JPEG images to the client.

To analyze these parts, we will see how the server side is built in the XML Grapher example.

The XML Grapher Server Side

In the XML Grapher example, a graph of nodes and links is displayed on top of a map. This ILOG JViews application is defined in the file `XmlGrapher.java`, located in:

```
<installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher
  /XmlGrapher.java
```

> ***Note:*** *We will not explain this part of the example in detail since it contains only standard ILOG JViews code and because the application on the server side really depends on the type of information you want to display. We will only see how this class is used to create the example.*

The XmlGrapher Class

The `XmlGrapher` class is a simple subclass of the ILOG JViews `IlvManagerView` class.

The main functionality of this small component is to read an XML file describing nodes and links and to create an ILOG JViews grapher that represents those nodes and links on top of a map. This is done in the method:

```
   public void setNetwork(URL url)
```

The XML files contain information on the map, the bitmap file of the map, and the projection. It contains a list of nodes, including the latitude and longitude of each node and information on links. The `setNetwork` method parses the XML file, creates the map, and places the nodes and the links on top of the map. It also applies an orthogonal link layout algorithm to automatically lay out the links.

You can look at some XML example files in:

```
<installdir>/demos/servlet/xmlgrapher/webpages/data
```

Creating the Servlet

Once the application is built, you need to create a servlet that produces images of the application to a client. ILOG JViews Graphics Framework provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvManagerServlet`. This class can be found in the package `ilog.views.servlet`.

In this section we will see a very simple servlet created for the XML Grapher example. To understand in depth how the servlet is working, you can read *The IlvManagerServlet Class* on page 205.

The servlet for the XML Grapher example is located in the file:

```
<installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher
  /servlet/XmlGrapherServlet.java
```

```
import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;

import ilog.views.*;
import ilog.views.servlet.*;

import demo.xmlgrapher.*;

public class XmlGrapherServlet extends IlvManagerServlet
{
  private XmlGrapher xmlGrapher;

  /**
   * Initializes the servlet.
   */
  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    xmlGrapher = new XmlGrapher();
    String xmlfile = config.getInitParameter("xmlfile");

    if (xmlfile == null) {
      xmlfile = config.getServletContext().getRealPath("/data/world.xml");
      xmlfile = "file:" + xmlfile;
    }
    try {
      xmlGrapher.setNetwork(new URL(xmlfile));
    } catch (MalformedURLException ex) {
    }
    setVerbose(true);
  }

  public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    return xmlGrapher;
  }

  protected float getMaxZoomLevel(HttpServletRequest request,
                                  IlvManagerView view)
  {
    return 30;
  }

}
```

As you see, the servlet class is very simple.

The `import` statements:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

are required to use the Java Servlet API.

The `import` statements:

```
import ilog.views.*;
import ilog.views.servlet.*;
```

are required for using ILOG JViews and the ILOG JViews servlet support.

The import statement:

```
import demo.xmlgrapher.*;
```

is required for the XML Grapher class.

The IlvManagerServlet class is an abstract Java class subclass of the HTTPServlet class from the Java servlet API. Our XmlGrapherServlet inherits from the IlvManagerServlet class and defines only three methods:

- The init Method
- The getManagerView Method
- The getMaxZoomLevel Method

The init Method

This method initializes the servlet by creating an XmlGrapher object:

```
public void init(ServletConfig config) throws ServletException
{
    xmlGrapher = new XmlGrapher();
    ...
```

Then an XML file is read by the XmlGrapher object using the setNetwork method:

```
String xmlfile = config.getInitParameter("xmlfile");
if (xmlfile == null)
  xmlfile
       = config.getServletContext().
              getRealPath("/data/world.xml");

try {
  xmlGrapher.setNetwork(new URL("file:" + xmlfile));
} catch (MalformedURLException ex) {
}
```

The XML file can be specified in the configuration of the servlet. By default, the file world.xml is used.

The getManagerView Method

This method is the only abstract method of the IlvManagerServlet class and should return an IlvManagerView that will be used to generate the image. Here we simply return the XmlGrapher object.

```
public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    return xmlGrapher;
  }
```

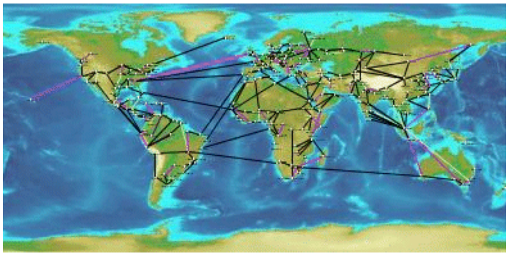
The getMaxZoomLevel Method

This method allows you to fix the user's maximum zoom level on the client side. Here we overwrite the method to return a larger value.

As you have seen, creating the servlet is very simple. This servlet can now answer HTTP requests from a client by sending JPEG images. If you have installed the example, you can try the following HTTP request:

```
http://localhost:8080/xmlgrapher/
demo.xmlgrapher.servlet.XmlGrapherServlet?request=image
            &format=JPEG&bbox=0,0,512,512
            &width=400
            &height=200
            &layer=Cities,Links,background%20Map
```

This produces the following image:



In this request, we ask the servlet named `demo.xmlgrapher.servlet.XmlGrapherServlet` to produce an image of size 400 x 200 showing the area (0, 0, 512, 512) of the manager with the layers “Cities,” “Links,” and “background Map” visible.

In most cases you do not have to know the servlet parameters because the Dynamic HTML objects or the Java classes provided by ILOG JViews for the client side will take care of the HTTP requests for you.

This example is a very simple servlet. This servlet is using the same `IlvManagerView` for all clients; this means that every client will see the same data. For more complex usage of the `IlvManagerServlet` classes, you can read *The IlvManagerServlet Class* on page 205.

Developing the Client Side

After creating the server side, you can create the client side. The ILOG JViews thin-client support allows you to easily build two different types of client:

- *Developing a Dynamic HTML Client* describes how to develop a client based on Dynamic HTML, which will run on Web browsers supporting Dynamic HTML. You build an HTML page for the DHTML client using predefined JavaScript components.
- *Developing a Thin-Java Applet Client* describes how to develop a thin-Java client that will run on Web browsers supporting Java 1.1. You build the thin Java client using predefined JavaBeans.

Both the Dynamic HTML client and the Java client provide the same set of predefined components, have the same type of functionality, and work with the same servlet on the server side. You choose between one client or the other depending on your deployment constraints.

Developing a Dynamic HTML Client

The static nature of HTML limits the interactivity of Web pages. Dynamic HTML allows you to create more interactive and engaging Web pages. It gives content providers new controls and allows them to manipulate the contents of HTML pages through scripting. To learn more about Dynamic HTML, you can visit the following Web sites:

- The Microsoft Web Workshop:

  `http://msdn.microsoft.com/workshop/`

- The Netscape DevEdge pages about DHTML:

  `http://developer.netscape.com/tech/dynhtml/index.html`

ILOG JViews provides a set of Dynamic HTML components written in JavaScript that allows you to build your DHTML pages very easily. The JavaScript files are located in `<installdir>/classes/thinclient/javascript`.

> ***Important:** Only the following releases of Web browsers will support DHTML: Netscape Communicator 4.x (on Windows and UNIX platforms) and Internet Explorer 5.0 and higher (on Windows platforms).*

The full reference documentation of each component can be found in the Dynamic HTML Component Reference located in:

`<installdir>/doc/jscript/index.html`

The XML Grapher Client Side with DHTML

We will now create a Dynamic HTML client for the XML Grapher example, starting with a very simple example and including most of the DHTML components. The full HTML file for the XML Grapher example is located in:

`<installdir>/demos/servlet/xmlgrapher/webpages/dhtml/index.html`

The IlvView JavaScript Component

The `IlvView` component (located in the `IlvView.js` file) is the main component. This component queries the servlet and displays the resulting image.

For including the JavaScript files:

- To use this component, you need to include the `IlvUtil.js` file , the `IlvView.js` file, the files for the superclasses of `IlvView`: `IlvAbstractView.js`, `IlvResizableView.js`, and `IlvEmptyView.js` as well as `IlvGlassView.js`.
- Instead of including the individual `.js` files of each component, you can add the file `framework.js` that is located in `<installdir>/classes/thinclient/framework/framework.js`. This file is a concatenation of all the `.js` files required for doing DHML thin client in the Graphics Framework.

Here is a simple HTML page that creates an `IlvView`:

```
<html>
<head>
<META HTTP-EQUIV="Expires" CONTENT="Mon, 01 Jan 1990 00:00:01 GMT">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
</head>
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvView.js"></script>
<script TYPE="text/javascript">

function init() {
  view.init()
  return false
}

function handleResize() {
  if (document.layers)
   window.location.reload()
}
</script>
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440
```

```
// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.toHTML()

</script>
</body>
</hmtl>
```

In this example we start by importing some JavaScript files:

```
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvView.js"></script>
```

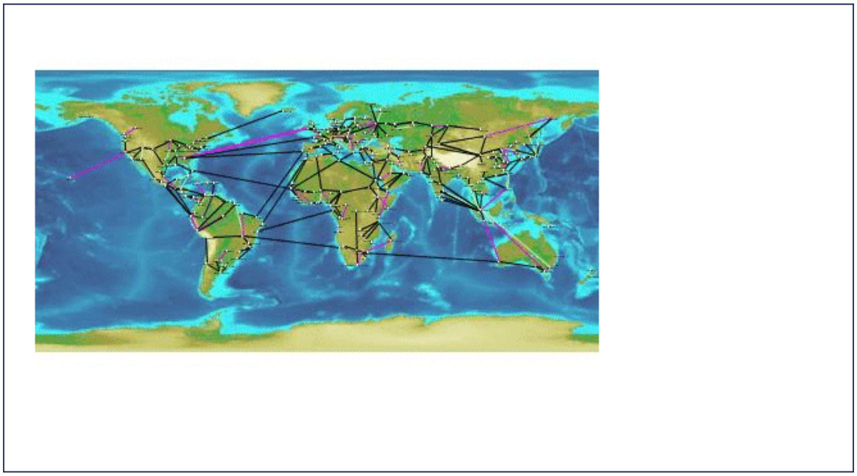
In the body of the page, we create an `IlvView` located in (40, 40) on the HTML page. The size is 440 x 270. This view displays images produced by the servlet `XmlGrapherServlet`. Note the `toHTML` method that creates the HTML necessary for the component.

This example also defines two JavaScript functions:

- The `init` function, called on the `onload` event of the page, initializes the `IlvView` by calling its `init` method.
- The `handleResize` function, called on the `onresize` event of the page, will reload the page if the browser is Netscape Communicator 4 or higher. This is necessary for a correct resizing of Dynamic HTML content on Communicator.

> ***Note:*** *The global* `ilvDispose` *function must be called in the* `onunload` *event of the HTML page. This function disposes of all resources acquired by the JViews DHTML components.*

Once the image is loaded from the server, the page now looks like this:



The IlvOverview JavaScript Component

The `IlvOverview` component (located in the `IlvOverview.js`) file shows an overview of the manager. An `IlvOverview` is linked to an `IlvView` component. By default, the `IlvOverview` queries the server to obtain an image of the global area and displays it. Once the overview is visible, a rectangle corresponding to the area visible in the main view is drawn on top of the overview. You can move this rectangle to change the area visible in the main view.

Here is the body of our previous example with an `IlvOverview`. Note that we cannot move the rectangle of the overview now because the complete area is visible in the main view. We will be able to do that later when we add the zooming functionality.

Note: *The lines added are in bold.*

```
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL(’/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet’)

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
```

```
overview.setColor('white')

view.toHTML()
overview.toHTML()

</script>
```

Compared to the previous example, we must add the new import statement for `IlvOverview.js`:

```
<script TYPE="text/javascript"  src="script/IlvOverview.js"></script>
```

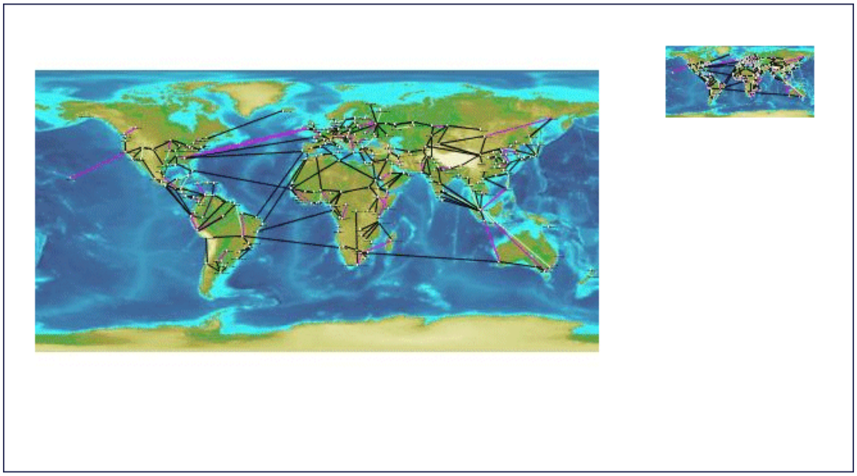
An `IlvOverview` object located in (x+w+50, y+4) with a size of 120 x 70 was created:

```
var overview = new IlvOverview(x+w+50, y+4, 120, 70, view)
```

The following line sets the color of the draggable rectangle:

```
overview.setColor('white')
```

The page looks now like this:



IlvLegend

We will now add an `IlvLegend` component to our page. The `IlvLegend` component shows a list of layers that are available on the server side, and allows you to turn the visibility of a layer on and off.

To use the `IlvLegend`, you must first include the `IlvLegend.js`.

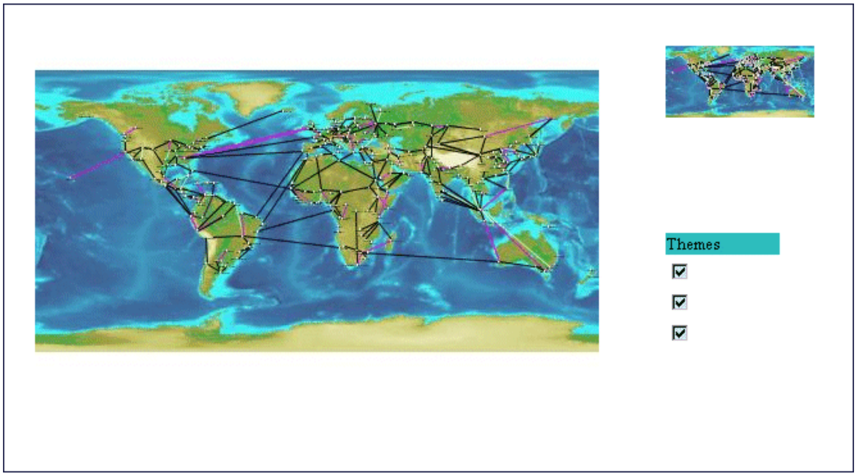
```
<script TYPE="text/javascript" src="IlvLegend.js"></script>
```

The body of the HTML file now looks like:

```
<body onload="init()" onunload="ilvDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
```

```
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor('white')

// Legend
var legend = new IlvLegend(x+w+50, y+150 ,120, 115, view)
legend.setTitle('Themes')
legend.setTitleBackgroundColor('#21bdbd')
legend.setTextColor('white')
legend.setBackgroundColor('#21d6d6')
legend.setTitleFontSize(2);

view.toHTML()
overview.toHTML()
legend.toHTML()
</script>
</body>
```

You should see the following page:



Themes

☑

☑

☑

The visibility of layers can now be turned on and off.

IlvButton

The `IlvButton` is a simple button that allows you to call some JavaScript code when clicking it. We will now add some buttons to our page to zoom in and out.

In addition to buttons, we will also add some Dynamic HTML panels to create a frame around our main view. A Dynamic HTML panel is an area of the page that can contain some HTML. Creating a panel is done using the class `IlvHTMLPanel`, defined in the `IlvUtil.js` file.

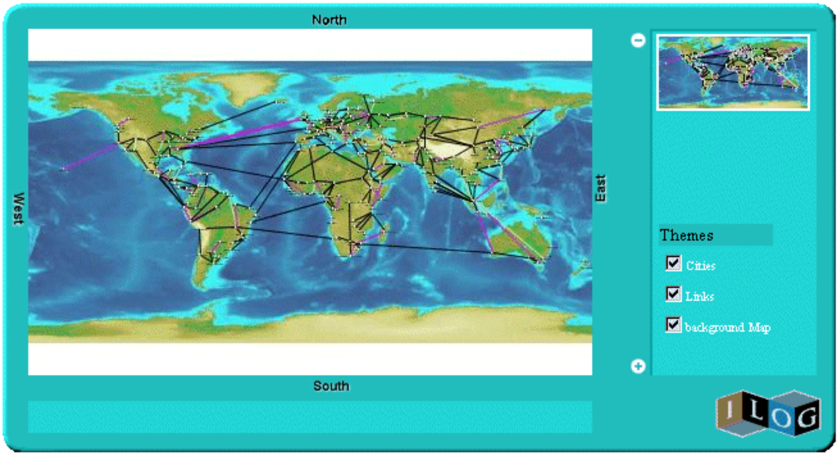
The body of the page is now:

```
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff" >
<script>

//position of the main view

var y = 40
var x = 40
var h = 270
var w = 440

// Creates a frame arround the main view
var frameBackground = new IlvHTMLPanel(’’)
frameBackground.setBounds(x-20, y-20, w+210, h+80)
frameBackground.setVisible(true)
frameBackground.setBackgroundColor(’#21bdbd’)

var frameTopLeft = new IlvHTMLPanel(’<IMG src="images/frame_topleft.gif">’)
frameTopLeft.setBounds(x-20, y-20, 40, 40)
frameTopLeft.setVisible(true)

var frameBottomLeft =new IlvHTMLPanel(’<IMG src="images/
frame_bottomleft.gif">’)
frameBottomLeft.setBounds(x-20, y+h+20, 40, 40)
frameBottomLeft.setVisible(true)

var frameTopRight = new IlvHTMLPanel(’<IMG src="images/frame_topright.gif">’)
frameTopRight.setBounds(x+w+150, y-20, 40, 40)
frameTopRight.setVisible(true)

var frameBottomRight = new IlvHTMLPanel(’<IMG src="images/
frame_bottomright.gif">’)
frameBottomRight.setBounds(x+w+150, y+h+20, 40, 40)
frameBottomRight.setVisible(true)

var frameTop = new IlvHTMLPanel(’<IMG src="images/frame_top.gif">’)
frameTop.setBounds(x+20, y-20, 570, 40)
frameTop.setVisible(true)

var frameBottom = new IlvHTMLPanel(’<IMG src="images/frame_bottom.gif">’)
frameBottom.setBounds(x+20, y+h+20, 570, 40)
frameBottom.setVisible(true)
```

```
var frameLeft = new IlvHTMLPanel('<IMG src="images/frame_left.gif">')
frameLeft.setBounds(x-20, y+20, 5, 270)
frameLeft.setVisible(true)

var frameRight = new IlvHTMLPanel('<IMG src="images/frame_right.gif">')
frameRight.setBounds(x+w+185, y+20, 5, 270)
frameRight.setVisible(true)

var border = new IlvHTMLPanel('')
border.setBounds(x+w+45, y, 130, h)
border.setVisible(true)
border.setBackgroundColor('#09a5a5')

var secondBorder = new IlvHTMLPanel('')
secondBorder.setBounds(x+w+47, y+2, 128, h-2)
secondBorder.setVisible(true)
secondBorder.setBackgroundColor('#21d6d6')

// message panel
var messagePanel = new IlvHTMLPanel('')
messagePanel.setBounds(x, y+h+20, w, 25)
messagePanel.setVisible(true)
messagePanel.setBackgroundColor('#21d6d6')
IlvButton.defaultMessagePanel = messagePanel;

// ILOG logo
var logo = new IlvHTMLPanel('<IMG src="images/ilog.gif">')
logo.setBounds(x+w+95, y+h+10, 85, 40)
logo.setVisible(true)

IlvButton.defaultInfoPanel = messagePanel;

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.setMessagePanel(messagePanel)

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor('white')
overview.setMessagePanel(messagePanel)

// Legend
var legend = new IlvLegend(x+w+50, y+150 ,120, 115, view)
legend.setTitle('Themes')
legend.setTitleBackgroundColor('#21bdbd')
legend.setTextColor('white')
legend.setBackgroundColor('#21d6d6')
legend.setTitleFontSize(2);

// Some buttons for navigation
```

```
var topbutton, bottombutton, rightbutton, leftbutton

topbutton = new IlvButton(x+w/2, y-15, 30, 13,'images/
north.gif','view.panNorth()')
topbutton.setRolloverImage('images/northh.gif')
topbutton.setToolTipText('pan north')
topbutton.setMessage('pan the map to the north')

bottombutton = new IlvButton(x+w/2, y+h, 33, 13,'images/
south.gif','view.panSouth()')
bottombutton.setRolloverImage('images/southh.gif')
bottombutton.setToolTipText('pan south')
bottombutton.setMessage('pan the map to the south')

leftbutton=new IlvButton(x-13, y+h/2-10, 13, 30,'images/
west.gif','view.panWest()')
leftbutton.setRolloverImage('images/westh.gif')
leftbutton.setToolTipText('pan west')
leftbutton.setMessage('pan the map to the west')

rightbutton=new IlvButton(x+w, y+h/2-25, 13, 28, 'images/east.gif',
'view.panEast()')
rightbutton.setRolloverImage('images/easth.gif')
rightbutton.setToolTipText('pan east')
rightbutton.setMessage('pan the map to the east')

// Buttons to zoom in and out
var zoominbutton, zoomoutbutton

zoominbutton=new IlvButton(x+w+30, y+h-16,12, 12, 'images/zoom.gif',
'view.zoomIn()')
zoominbutton.setRolloverImage('images/zoomh.gif')
zoominbutton.setMessage('click to zoom by 2')
zoominbutton.setToolTipText('Zoom In')

zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, 'images/unzoom.gif',
'view.zoomOut()')
zoomoutbutton.setRolloverImage('images/unzoomh.gif')
zoomoutbutton.setMessage('click to zoom out by 2')
zoomoutbutton.setToolTipText('Zoom Out')

view.toHTML()
overview.toHTML()
legend.toHTML()
topbutton.toHTML()
bottombutton.toHTML()
leftbutton.toHTML()
rightbutton.toHTML()
zoomoutbutton.toHTML()
zoominbutton.toHTML()

</script>
</body>
</hmtl>
```

The page now looks like this:



A frame around the page was created by the following lines:

```
var frameBackground = new IlvHTMLPanel(’’)
frameBackground.setBounds(x-20, y-20, w+210, h+80)
frameBackground.setVisible(true)
frameBackground.setBackgroundColor(’#21bdbd’)

var frameTopLeft = new IlvHTMLPanel(’<IMG src="images/frame_topleft.gif">’)
frameTopLeft.setBounds(x-20, y-20, 40, 40)
frameTopLeft.setVisible(true)

var frameBottomLeft=new IlvHTMLPanel(’<IMG src="images/frame_bottomleft.gif">’)
frameBottomLeft.setBounds(x-20, y+h+20, 40, 40)
frameBottomLeft.setVisible(true)

var frameTopRight = new IlvHTMLPanel(’<IMG src="images/frame_topright.gif">’)
frameTopRight.setBounds(x+w+150, y-20, 40, 40)
frameTopRight.setVisible(true)

var frameBottomRight = new IlvHTMLPanel(’<IMG src="images/
frame_bottomright.gif">’)
frameBottomRight.setBounds(x+w+150, y+h+20, 40, 40)
frameBottomRight.setVisible(true)

var frameTop = new IlvHTMLPanel(’<IMG src="images/frame_top.gif">’)
frameTop.setBounds(x+20, y-20, 570, 40)
frameTop.setVisible(true)

var frameBottom = new IlvHTMLPanel(’<IMG src="images/frame_bottom.gif">’)
frameBottom.setBounds(x+20, y+h+20, 570, 40)
frameBottom.setVisible(true)
```

```
var frameLeft = new IlvHTMLPanel(’<IMG src="images/frame_left.gif">’)
frameLeft.setBounds(x-20, y+20, 5, 270)
frameLeft.setVisible(true)

var frameRight = new IlvHTMLPanel(’<IMG src="images/frame_right.gif">’)
frameRight.setBounds(x+w+185, y+20, 5, 270)
frameRight.setVisible(true)
```

This creates four DHTML panels for the corners, four additional panels for the sides, and a panel for the background. The corners and the sides of the frame are composed of simple GIF images.

Four buttons to pan south, north, east, and west have been added by the lines:

```
topbutton = new IlvButton(x+w/2, y-15, 30, 13,’images/
north.gif’,’view.panNorth()’)
topbutton.setRolloverImage(’images/northh.gif’)
topbutton.setToolTipText(’pan north’)
topbutton.setMessage(’pan the map to the north’)

bottombutton = new IlvButton(x+w/2, y+h, 33, 13,’images/
south.gif’,’view.panSouth()’)
bottombutton.setRolloverImage(’images/southh.gif’)
bottombutton.setToolTipText(’pan south’)
bottombutton.setMessage(’pan the map to the south’)

leftbutton=new IlvButton(x-13, y+h/2-10, 13, 30,’images/
west.gif’,’view.panWest()’)
leftbutton.setRolloverImage(’images/westh.gif’)
leftbutton.setToolTipText(’pan west’)
leftbutton.setMessage(’pan the map to the west’)

rightbutton=new IlvButton(x+w, y+h/2-25, 13, 28, ’images/east.gif’,
’view.panEast()’)
rightbutton.setRolloverImage(’images/easth.gif’)
rightbutton.setToolTipText(’pan east’)
rightbutton.setMessage(’pan the map to the east’)
```

A button is defined by its position and size, two images, the main image and the rollover image, and a piece of JavaScript to be executed when the button is clicked.

Note that in order to pan to the north, we simply use the `panNorth` method of `IlvView`.

Two additional buttons have been created to zoom in and out, by the lines:

```
var zoominbutton, zoomoutbutton

zoominbutton=new IlvButton(x+w+30, y+h-16,12, 12, ’images/zoom.gif’,
’view.zoomIn()’)
zoominbutton.setRolloverImage(’images/zoomh.gif’)
zoominbutton.setMessage(’click to zoom by 2’)
zoominbutton.setToolTipText(’Zoom In’)

zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, ’images/unzoom.gif’,
’view.zoomOut()’)
```

```
zoomoutbutton.setRolloverImage(’images/unzoomh.gif’)
zoomoutbutton.setMessage(’click to zoom out by 2’)
zoomoutbutton.setToolTipText(’Zoom Out’)
```

You have noted that each button has a `message` property. The message will be automatically displayed in the status window of the browser when the mouse is over the button. The message can also be displayed in an additional panel. That is why the line:

```
IlvButton.defaultInfoPanel=messagePanel
```

tells us that messages of buttons will also be displayed in the DHTML message panel.

IlvZoomTool

The next step is to add an `IlvZoomTool`. The `IlvZoomTool` is a DHTML component that shows a set of buttons. Each button corresponds to a zoom level; clicking the button will zoom the view to this zoom level. The button corresponding to the current zoom level is visually different from others so that you can tell what the current zoom level is. The component can be vertical or horizontal, and the images of the buttons can be customized.

To add the component, we add the following lines to the page:

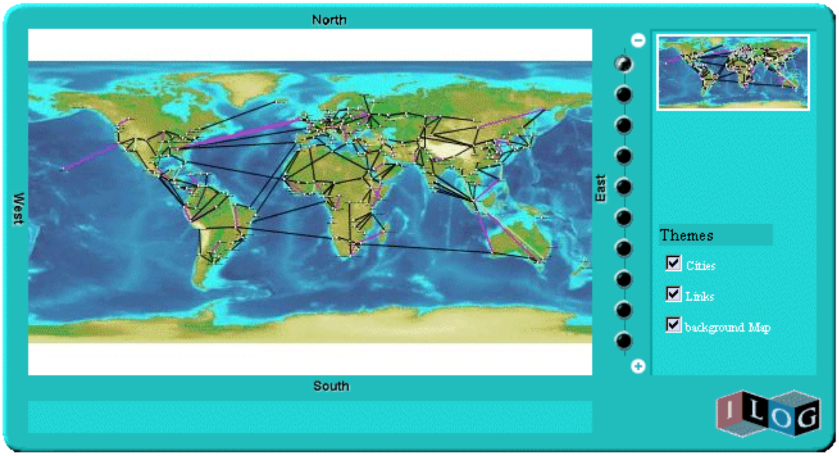
```
<script TYPE="text/javascript" src="script/IlvZoomTool.js"></script>
```

to import the script.

Note that this component uses the `IlvButton` class, so the `IlvButton.js` script must be included also.

```
var zoomtool = new IlvZoomTool(x+w+25, y+15, 25, h-30, 10 , view)
zoomtool.setOrientation(’Vertical’)
zoomtool.upImage = ’images/button.gif’
zoomtool.rolloverUpImage = ’images/buttonh.gif’
zoomtool.downImage = ’images/button.gif’
zoomtool.rolloverDownImage = ’images/buttonh.gif’
zoomtool.currentImage = ’images/center.gif’
zoomtool.rolloverCurrentImage = ’images/centerh.gif’

zommtool.toHTML()
```

The page now looks like this, with the vertical zoom tool on the right of the main view:



IlvZoomInteractor

Until now we have added components and buttons to our page. We will now add an interactor that allows direct interaction with the image. The `IlvZoomInteractor` allows the user to select an area on the image to zoom this area. Installing an interactor on the view is simple: you need only create the interactor and set it to the view:

```
var zoomInteractor = new IlvZoomInteractor()
view.setInteractor(zoomInteractor)
```

In our example, we add a button that will install the interactor. We add the following lines to the page:

```
<script TYPE="text/javascript"
      src="script/IlvInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvDragRectangleInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvZoomInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvInteractorButton.js"></script>
```

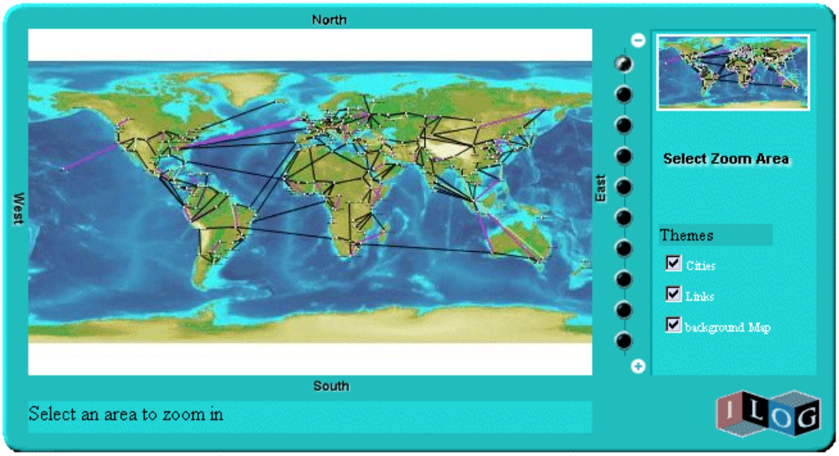
To use the interactor, we have to import three JavaScript files: `IlvInteractor.js`, `IlvDragRectangleInteractor.js`, and `IlvZoomInteractor.js`. This is because the `IlvZoomInteractor` component is a subclass of the `IlvDragRectangleInteractor` component.

Then we add the following lines to the body of the page:

```
var zoomInteractor = new IlvZoomInteractor()
zoomInteractor.setLineWidth(1)
zoomInteractor.setColor('#00ffff')

...
```

```
var zoomrectbutton

zoomrectbutton=new IlvInteractorButton(x+w+50, y+90, 112, 24,
                                       'images/zoomrect.gif', zoomInteractor, view)

zoomrectbutton.setRolloverImage('images/zoomrecth.gif')
zoomrectbutton.setMessage('click to set zoom mode')
zoomrectbutton.setToolTipText('Zoom Mode')

...

zoomrectbutton.toHTML()
```

This results in the following page:



☑ Cities
☑ Links
☑ background Map

Select an area to zoom in

You can now click the “Select Zoom Area” button to install the interactor and then select an area to zoom in.

IlvPanInteractor

The `IlvPanInteractor` is another interactor that allows the user to click in the main view to pan the view. Just like the `IlvZoomInteractor`, we use the `setInteractor` method of `IlvView` to install the interactor. In our example, we add another button that will install this interactor, just as was done for *IlvZoomInteractor* on page 192. We will now be able to switch from the “Pan” mode and the “Zoom” mode.

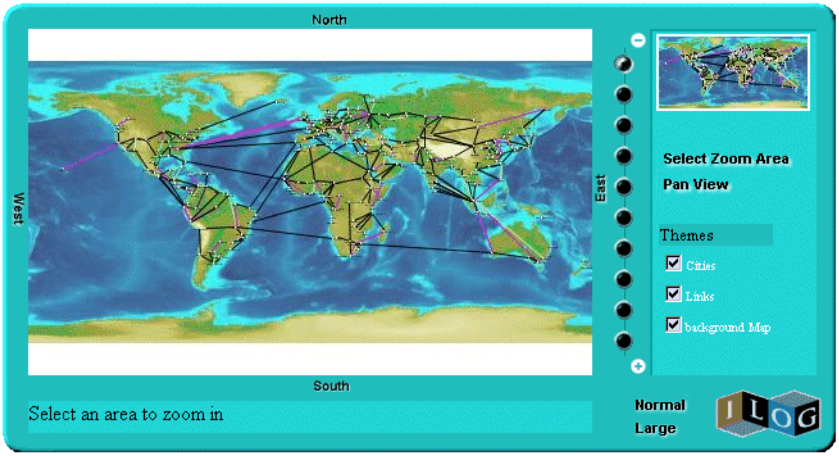
To be able to use the component, we import the corresponding JavaScript file:

```
<script TYPE="text/javascript"
      src="script/IlvPanInteractor.js"></script>
```

Then we add the following lines to the body of the page:

```
var panInteractor  = new IlvPanInteractor()
panbutton=new IlvInteractorButton(x+w+50, y+110, 63, 22, 'images/pan.gif',
                                  panInteractor, view)
panbutton.setRolloverImage('images/panh.gif')
panbutton.setMessage('click to set pan mode')
panbutton.setToolTipText('Pan Mode')
...

panbutton.toHTML()
```

The page now has one additional button labelled “Pan View”:



Select an area to zoom in

The example is now complete, we have used most of the DHTML components provided by ILOG JViews.

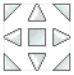
IlvPanTool

The `IlvPanTool` component (located in the `IlvPanTool.js` file) is a component that allows panning the view in all directions. You create the component in this way:

```
var pantool = new IlvPanTool(10, 10, view)
pantool.toHTML()
```

Note that this component uses the `IlvButton` class, so the `IlvButton.js` script must be included also.

This component looks like this:



IlvMapInteractor and IlvMapRectInteractor

The `IlvMapInteractor` and `IlvMapRectInteractor` are two additional interactors that can be used to perform an action on the server side when a point or an area of the image is selected by the client. These interactors and how to use them are described in detail in *Adding Client/Server Interactions* on page 201.

Developing a Thin-Java Applet Client

The ILOG JViews thin-client support provides two types of predefined clients: the DHTML client (see *Developing a Dynamic HTML Client* on page 180) and the thin Java client that is described here.

With the ILOG JViews thin-client support, you can develop the client side using the thin Java client classes to create very light applets. To build the thin Java client, we will use predefined JavaBeans.

Table 10.1 lists the JavaBeans provided to build the thin-Java client. The classes are defined in the package `ilog.views.thinclient`.

Table 10.1 *Thin-Client Java Beans*

Java Class	Description
`IlvView`	The main JavaBean that shows the image resulting from the servlet.
`IlvOverview`	A JavaBean that shows an overview of the main view and allows panning.
`IlvLegend`	A JavaBean that shows all the layers available and allows you to turn the visibility of a layer on or off.
`IlvPanTool`	A JavaBean that allows panning the main view.
`IlvZoomTool`	A JavaBean that allows zooming the main view.
`IlvZoomInteractor`	An interactor that lets you drag a rectangle to select an area for zooming.
`IlvPanInteractor`	An interactor that lets you pan the main view by dragging the image.
`IlvDragRectangleInteractor`	An interactor that lets you drag a rectangle in the main view.
`IlvMapInteractor`	An interactor that lets you click the main view to perform an action on the server.

Table 10.1 Thin-Client Java Beans (Continued)

Java Class	Description
`IlvMapRectInteractor`	An interactor that lets you select an area in the main view to perform an action on the server.
`IlvImageButton`	A simple button with image.

The XML Grapher Client Side with Java

You can use your favorite IDE to create your thin-Java client. We explain here the applet provided in the XML Grapher example.

The source code of the applet is located in:

```
<installdir>/demos/servlet/xmlgrapher/src/demo/xmlgrapher/applet
  /XmlGrapherApplet.java
```

The HTML page containing the applet is located in:

```
<installdir>/demos/servlet/xmlgrapher/webpages/applet/index.html
```

HTML for the Thin-Java Client Example

First we will take a look at the HTML page:

```
<html>
<head>
<title>JViews Servlet (applet) @ ILOG</title>
</head>
<body BGCOLOR="white">
<applet archive="jviewsthin.jar,xmlgrapher.jar"
        code=demo.xmlgrapher.applet.XmlGrapherApplet.class
        WIDTH=650 HEIGHT=350>
  <PARAM NAME="servlet"
   VALUE=
"http://localhost:8080/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet">
</applet>
</body>
</html>
```

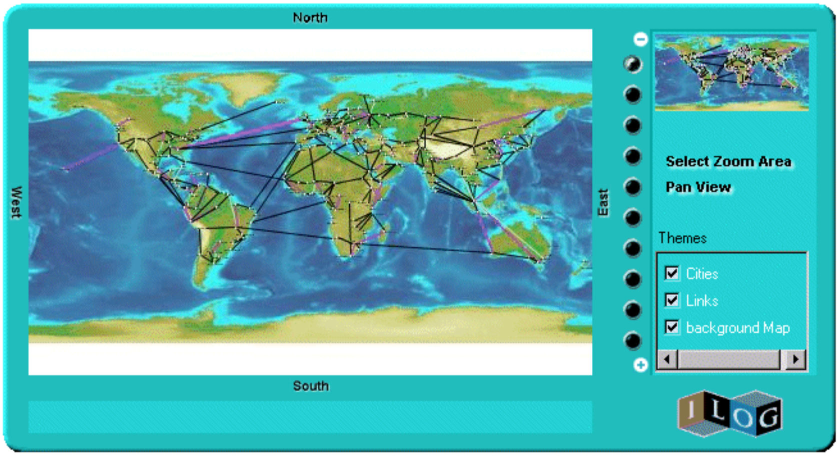
This page is using two JAR files: `xmlgrapher.jar`, the JAR file containing the applet, and `jviewsthin.jar`. The `jviewsthin.jar` JAR file is the JAR file containing only the classes required to create the thin-client applet. It contains the `ilog.views.thinclient` package.

The HTML page also defines a parameter servlet for the applet. This allows you to easily change the location of the servlet without recompiling the applet.

If the TOMCAT server is running, you can see the applet by opening the page:

```
http://localhost:8080/xmlgrapher/applet/index.html
```

You will see this page:



Importing the Package

The `XmlGrapherApplet.java` file starts by importing the necessary packages, in particular the `ilog.views.thinclient` package.

```
import ilog.views.thinclient.*;
```

We also import various packages necessary to build the AWT user interface.

```
import java.awt.*;
import java.io.*;
import java.net.*;
import java.awt.event.*;
import java.applet.*;
```

IlvView JavaBean

This component corresponds to the `IlvView` JavaScript component and has the same functionality. It is the main JavaBean that queries the servlet and displays the images.

This component is created in the `init` method of the applet by the code:

```
view = new IlvView();
try {
  view.setWaitingImage(new URL(getDocumentBase(), "images/wait.gif"));
} catch (Exception ex) {}
view.setBounds(20, 20, bounds.width - 210, bounds.height - 80);
view.setRequestURL(getParameter("servlet"));
view.setBackground(Color.white);
add(view);
```

The most important parameter is the path to the servlet:

```
view.setRequestURL(getParameter("servlet"));
```

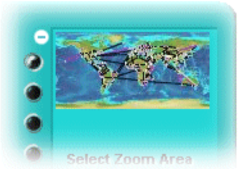
You see that the path to the servlet is taken from the parameter of the APPLET tag, as we discussed in *HTML for the Thin-Java Client Example* on page 196.

We also set the image displayed when the users must wait for a server answer:

```
view.setWaitingImage(new URL(getDocumentBase(), "images/wait.gif"));
```

IlvOverview JavaBean

The `IlvOverview` JavaBeans shows an overview of the full area of the manager and allows panning the main view (`IlvView`) by dragging a rectangle showing the area currently visible on the main view.



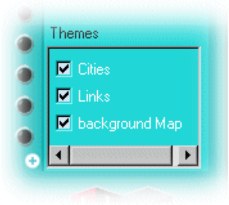
This component is created by the following code:

```
IlvOverview overview = new IlvOverview();
overview.setView(view);
overview.setForeground(Color.white);
overview.setBounds(2, 2, 120, 70);
```

You have noted that the overview is attached to the main view by using the `setView` method.

IlvLegend JavaBean

The `IlvLegend` JavaBean shows the list of layers available on the server side as a list of check boxes that you can turn on or off.



☑ Cities
☑ Links
☑ background Map

Once again this component must be attached to the main view using its `setView` method. Here is the piece of code that creates this component.

```
IlvLegend legend = new IlvLegend();
legend.setView(view);
legend.setForeground(Color.white);
```

IlvImageButton

This component is a simple button that can display an image instead of a simple text as AWT buttons. Its is used in the XmlGrapherApplet.java code to create various buttons. Here is the code that creates the “North” navigation button:

```
panNorthButton = createButton(viewBounds.x +
                              viewBounds.width/2 - 15,
                              viewBounds.y - 15, 30, 13,
                              "north.gif", "northh.gif",
                              "pan the map to the north");
add(panNorthButton);
```

Where the createButton method is the following:

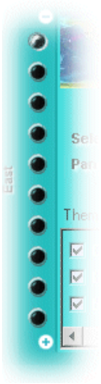
```
private IlvImageButton createButton(int x, int y, int w, int h,
                                    String image,
                                    String rolloverImage,
                                    String message)
{
  IlvImageButton button = new IlvImageButton();
  button.setBounds(x, y, w, h);
  button.setImage(getImage(image));
  button.setRolloverImage(getImage(rolloverImage));
  button.addActionListener(this);
  button.setPaintingBorder(false);
  button.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
  button.setMessageComponent(messageLabel);
  button.setMessage(message);
  return button;
}
```

Two images can be defined, one for the normal view of the button and another one used to displayed the button when the mouse is over the button.

You have noted that a message can be installed on the button. This message will be displayed in the status window of the browser but can also be displayed in another AWT component. This is done by the setMessageComponent method.

IlvZoomTool JavaBean

The IlvZoomTool is a JavaBean component that shows a set of buttons. Each button corresponds to a zoom level. Clicking the button will zoom the view to this zoom level. The button corresponding to the current zoom level is visually different from others so that you can tell what the current zoom is. The component can be vertical or horizontal, and the images of the buttons can be customized.



Here is the code that creates the `IlvZoomTool` in our example:

```
IlvZoomTool zoomtool = new IlvZoomTool();
zoomtool.setView(view);
zoomtool.setZoomInImage(getImage("button.gif"));
zoomtool.setZoomOutImage(getImage("button.gif"));
zoomtool.setZoomInRolloverImage(getImage("buttonh.gif"));
zoomtool.setZoomOutRolloverImage(getImage("buttonh.gif"));
zoomtool.setCurrentZoomImage(getImage("center.gif"));
zoomtool.setCurrentZoomRolloverImage(getImage("centerh.gif"));
zoomtool.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
zoomtool.setBounds(viewBounds.x + viewBounds.width + 20,
                   viewBounds.y + 15, 25, viewBounds.height - 30);
zoomtool.setOrientation(IlvZoomTool.VERTICAL);
```

Once again the component is associated with the main `IlvView` using its `setView` method. We are also using various methods to change the default images used by the `IlvZoomTool`.

IlvZoomInteractor

The `IlvZoomInteractor` is an interactor that allows the user to select an area on the image in order to zoom to this area. Installing an interactor on the view is simple: You need only create the interactor and set it to the view:

```
var zoomInteractor = new IlvZoomInteractor()
view.setInteractor(zoomInteractor)
```

In our example, we add a button that will install the interactor. The creation of the `IlvZoomInteractor` is done the following way:

```
zoomInteractor = new IlvZoomInteractor();
zoomInteractor.setLineWidth(1);
zoomInteractor.setColor(Color.white);
```

The interactor is installed on the view later.

IlvPanInteractor

The `IlvPanInteractor` is another interactor that allows the user to click in the main view to pan the view. Just like the `IlvZoomInteractor`, we use the `setInteractor` method of `IlvView` to install the interactor. In our example, we add another button that will install this interactor, just as was done for *IlvZoomInteractor* on page 200. We will then be able to switch between the "Pan" mode and the "Zoom" mode.

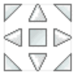
The component is simply created by the line:

```
panInteractor = new IlvPanInteractor();
```

IlvPanTool JavaBean

The `IlvPanTool` JavaBean is a component that allows you to pan the view in all directions. Note that this component is not used in our XML Grapher applet.

This component looks like this:



IlvMapInteractor and IlvMapRectInteractor

`IlvMapInteractor` and `IlvMapRectInteractor` are two additional interactors that can be used to perform an action on the server side when a point or an area of the image is selected by the client. These interactors and how to use them are described in detail in *Adding Client/Server Interactions* on page 201.

Adding Client/Server Interactions

The ILOG JViews thin-client support gives you a simplified way to define new actions that should take place on the server side. For example, suppose you want to allow the user to delete a graphic object that appears on the generated image. Part of this action—clicking the image to select the object—must be done on the client side. The destruction of the object must be done on the server side before a new image is generated. The notion of "server-side action" exists to perform such behavior. An action is defined by a name and a set of string parameters.

The Client Side

In a dynamic HTML client, you tell the server to perform an action using the `performAction` method of the `IlvView` JavaScript component.

Here is an example that asks the server side to execute the action “delete” with coordinate parameters, assuming that `view` is an `IlvView`:

```
var x = 100;
var y = 50;
var params = new Array();
params[0]=x;
params[1]=y;
view.performAction(“delete”, params);
```

In a thin-Java client the system is the same:

```
float x = 100f;
float y = 50f;
String[] params = new String[2];
params[0] = Float.toString(x);
params[1] = Float.toString(y);
view.performAction(“delete”, params);
```

The `performAction` method will ask the server for a new image. In the image request, additional parameters are added so that the server side can execute the action. Thus, the `performAction` call results in only one client/server round-trip.

Note that predefined interactors are provided to help you define new actions on the client side. They are explained in *Predefined Interactors* on page 203.

The Server Side

On the server side, we need to detect that an action was requested and execute the action. This is done using the interface `ilog.views.servlet.ServerActionListener`.

To be able to listen and execute an action on the server side, you simply add an action listener to your servlet. In the `performAction` method of the listener, you check the action name and perform the action.

For the “delete” action, we would add the following lines of code in the `init` method of the servlet:

```
addServerActionListener(new ServerActionListener() {
public void actionPerformed(ServerActionEvent e) throws ServletException
{
    if (e.getActionName().equals("delete")) {
      IlvPoint p = e.getPointParameter(0);
      // find object under this point and delete it if there is one.
    }
  }});
```

The `ServerActionEvent` object can give you all necessary information about the action, the name, and its parameters.

Predefined Interactors

Two predefined interactors are provided to help you create new actions: `IlvMapInteractor` and `IlvMapRectInteractor`.

`IlvMapInteractor` allows the user to click in the map; it will ask the server to execute an action, with the coordinates of the clicked point passed as parameters. The second interactor is almost the same except that the user selects an area of the image instead of clicking on it.

Generating a Client-Side Image Map

If you are creating a Dynamic HTML client, the ILOG JViews thin-client support allows you to create a client-side image map. Image maps are images with an attached map that points out certain hot spots, or clickable areas. In the ILOG JViews thin-client support, a clickable area can be generated for each graphic object of the manager. To create a client side image map, you will have two things to do:

- *The Server Side* – defining the image map on the server side.
- *The Client Side* – using the image map on the client side.

The Server Side

The servlet provided by ILOG JViews (`IlvManagerServlet`) is able to generate an image map for your ILOG JViews application, but it is likely that you do not want to generate a clickable area for every graphic object. On the server side, you will then have to tell the manager servlet which ILOG JViews layer and which graphic object are part of the image map generation. For both layer and graphic object, this is done by setting a property on them.

On a layer, assuming that the variable `manager` is an `IlvManager`, you will do:

```
manager.getManagerLayer(index).setProperty(
     IlvManagerServlet.ImageMapAreaGeneratorProperty, Boolean.TRUE);
```

On a graphic object you can do almost the same thing, but the value of the property must be an instance of the class `IlvImageMapAreaGenerator`. This class is responsible for generating the AREA part of the image map.

Note that the same instance of `IlvImageMapAreaGenerator` can be used for all graphic objects.

By default, `IlvImageMapAreaGenerator` will generate a rectangular area with no HREF in it. You will have to subclass it to generate an HREF for your graphic object.

10. Thin-Client Support

Here is an example that creates a custom `IlvImageMapAreaGenerator` and sets it on some objects:

```
IlvGraphic object1, object2;
....
IlvImageMapAreaGenerator generator = new IlvImageMapAreaGenerator() {
    public String generateHREF(IlvManagerView v, IlvGraphic obj) {
    String href;
    // place here code the
    // computes the URL depending on the graphic object
    return href;
  }

};

object1.setProperty(IlvManagerServlet.ImageMapAreaGeneratorProperty,
                      generator);
object2.setProperty(IlvManagerServlet.ImageMapAreaGeneratorProperty,
                      generator);
```

The HREF can be a URL to which the browser will jump when the area is clicked, but it can also be a call to a JavaScript method.

For example, in our XML Grapher example, we can define the generator this way:

```
IlvImageMapAreaGenerator generator = new IlvImageMapAreaGenerator() {

  public String generateALT(IlvManagerView v, IlvGraphic obj) {
     return ((GrapherNode)obj).getLabel();
  }

  public String generateHREF(IlvManagerView v, IlvGraphic obj) {
     return "javascript:doSomething('"+
             ((GrapherNode)obj).getLabel()+"')";
  }
};
```

In this example, the HREF generated is a call to the JavaScript method `doSomething`. You will have to define this method in the HTML page.

For more information about customizing an area, see the `IlvImageMapAreaGenerator` class in the *ILOG JViews Reference Manual*.

The Client Side

To tell the Dynamic HTML client to generate a client-side image map, you only need to set the `imageMap` property of the `IlvView` JavaScript component to `true`:

```
var view = new IlvView(40, 40, 300, 400);
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet');
view.setGenerateImageMap(true);
```

When this is done, the `IlvView` component will ask the servlet to generate the image map.

Lastly, to make the image map visible, there are two possibilities. You can:

- Directly call the `showImageMap` method of `IlvView`:

  ```
  view.showImageMap();
  ```

- Use the `IlvImageMapInteractor` class. This class is a simple interactor that will show the image map when installed and hide it when de-installed.

The IlvManagerServlet Class

Developing the server side of a thin-client ILOG JViews application consists of creating a servlet that can produce an image to the client. ILOG JViews provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvManagerServlet`. This class can be found in the package `ilog.views.servlet`.

The `IlvManagerServlet` class is an abstract Java subclass of the `HTTPServlet` class from the Java servlet API.

The Servlet Parameters

The servlet can respond to three different types of HTTP requests, the “image” request, the “image map” request, and the “capabilities” request. The image request will return an image from the ILOG JViews manager. The capabilities request will return information to the client, such as the layers available in the manager and the global area of the manager. This information allows the client to know the capabilities of the servlet in order to build the image request. When developing the client side of your application, you will use the DHTML scripts or the JavaBeans provided by ILOG JViews; both will create the HTTP request for you, so you do not really need to write the HTTP request yourself.

The Image Request

The image request produces a JPEG image from the manager. The request has the following syntax, assuming that `myservlet` is the name of the servlet:

```
http://host/myservlet?request=image
        &bbox=x,y,width,height (area in the manager coordinate
```

```
  system)
          &width=width of the returned image
          &height=height of the returned image
          &layer=comma separated list of layers
          &format=JPEG
          &bgcolor=0xFFFFFF
```

Here is a list of parameters and their meanings.

Table 10.2 *Parameters of the IlvManagerServlet*

Parameter Name	**Parameter Value**	**Description**
`request`	image	Asks the servlet to generate an image.
`bbox`	Float, Float, Float, Float	The area of the manager that will be displayed in the image. The first two values are the upper left corner of the area. The last two values are the width and height of the area.
`width`	Integer	Width of the resulting image.
`height`	Integer	Height of the resulting image.
`format`	JPEG	The format of the resulting image.
`layer`	Comma-separated list of strings. For example: Cities, Roads	The layers of the `IlvManager` that will be visible.
`bgcolor`	`0xrrggbb` For example, `0xffffff` for white	The background color of the resulting image. This parameter is optional.
`action`	actionName(param1, param2)	Specifies an action to be executed on the server before the image is generated.

The following request will produce a JPEG image of size (250, 250) showing the area (0, 0, 1000, 1000) of the manager; only the layers named “Cities” and “Roads” will be visible:

```
     http://host/myservlet?request=image
           &bbox=0,0,1000,1000
           &width=250
           &height=250
           &layer=Cities,Roads
           &format=JPEG
```

The Capabilities Request

The capabilities request produces information to the client. This request returns information on the manager.

The capabilities request has the following syntax

```
     http://host/myservlet?request=capabilities
           &format=(html|octet-stream)
         [ &onload= <a string> ]
```

The `request` parameter set to `capabilities` instead of `image` tells the servlet to return the capabilities information. The `format` parameter tells which format should be returned.

The result can be of two different formats, HTML or Octet stream.

HTML Format

The HTML format is used when the client is a Dynamic HTML client. In this case, the result is a empty HTML page that contains some JavaScript code. The JavaScript code is executed on the client side, and some information variables are then available.

```
<html>
<head>
<script language="JavaScript">
var minx=0.0;
var miny=0.0;
var maxx=1024.0;
var maxy=512.0;
var themes=new Array();
var overviewthemes=new Array();
themes[0]="a layer name";
overviewthemes[0]=true;
themes[1]="another layer";
overviewthemes[1]=true;
themes[2]="a third layer";
overviewthemes[2]=true;
var maxZoom=6;
</script>
</head>
<body>
</body>
</html>
```

The variables `minx`, `miny`, `maxx`, `maxy` are defining the global area of the manager that can be queried. The `themes` variable is the list of layers available on the server side. The overviewthemes variable tells if a layer should be visible in the overview window. The maxZoom variable is the maximum level of zoom the application should perform.

The onload parameter allows you to specify a String that is used for the onload event of the generated HTML page. When an onload parameter is specified, the body tag of the HTML page is the following:

```
<body onLoad="+onload+">
```

Octet-Stream Format

The octet-stream format is used when the client is a Java applet. In this case, the result is a stream of octets. The data is produced using a java.io.DataOutput and can be read using a java.io.DataInput. It is organized as follows:

```
Float: left coordinate of manager’s bounding box.
Float: top coordinate of manager’s bounding box.
Float: right coordinate of manager’s bounding box.
Float: bottom coordinate of manager’s bounding box.
Int: number of layers.

for each layer:
   String (UTF format): name of the layer.
   Boolean: is the layer an overview layer.

Float: Maximum zoom level
```

You see that this format gives the same type of information as the HTML format. Once again, you do not need to decode or read these formats. The client-side components provided by ILOG JViews will do that for you.

The Image Map Request

The image map request produces an image and a client-side image map. The parameters for this request are the same as for the image request except that the request parameter must have the value imagemap.

For example, the following code to the servlet:

```
http://host/myservlet?request=imagemap
      &width=400
      &height=200
      &bbox=0,0,500,500
      &format=JPEG
      &layer=Cities,Links,background%20Map
```

will produce something like:

```
<html>
<body>
<map name="imagemap">
<area shape="rect" coords="242,81,261,83" href="..." >
....
```

```
</map>
<img usemap="#imagemap" width="400" height="200"
src="myservlet?request=image&layer=Cities,Links,background%20Map&width=400&form
at=JPEG&bbox=0,0,500,500&height=200" border=0>
</body>
</html>
```

The call generates an HTML document containing the client-side image map and an image. The contents of the image are then generated by another call to the servlet.

The graphic objects that are taken into account when generating the map can be specified as well as the shape of the clickable area and what appends when you click on it. All this is explained in *Generating a Client-Side Image Map* on page 203.

The image map request has two additional optional parameters:

- The `mapname` parameter allows you to specify the name of the map. The default name is `imagemap`.
- The `onload` parameter allows you to specify a String that is used for the onload event of the generated HTML page. When an `onload` parameter is specified, the body tag of the HTML page is the following:

  ```
  <body onLoad="+onload+">
  ```

Multiple Sessions

The XML Grapher has presented a very simple example that creates a single manager view for the servlet. This means that all calls to the servlet (that is, all clients) are looking at the same view. This is fine when the same data is used for all clients. In some applications—for example, when you want to allow the user to edit the graphic representation—you might want to have a view (and thus a manager) for each client. In this case, you might use the notion of *HTTP sessions*. You can then create a view and a manager and store them as parameters of the session.

Here is a slightly modified version of the XML Grapher servlet using sessions:

```
package demo.xmlgrapher.servlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import ilog.views.*;
import ilog.views.servlet.*;
import demo.xmlgrapher.*;

public class XmlGrapherServlet extends IlvManagerServlet
{
  String xmlfile;

  public void init(ServletConfig config)
       throws ServletException
  {
```

```
    xmlfile = config.getInitParameter("xmlfile");
    if (xmlfile == null)
      xmlfile = config.getServletContext().
                   getRealPath("/data/world.xml");
    setVerbose(true);
  }

  protected void prepareSession(HttpServletRequest request)
  {
    HttpSession session = request.getSession();
    if (session.isNew()) {

      XmlGrapher xmlGrapher = new XmlGrapher();
      try {
        xmlGrapher.setNetwork(new URL("file:" + xmlfile));
      } catch (MalformedURLException ex) {
      }
      session.putValue("IlvManagerView", xmlGrapher);
    }
  }

  public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    HttpSession session = request.getSession(false);
    if (session!= null)
      return (IlvManagerView)session.getValue("IlvManagerView");
    else
      throw new ServletException("session problem");
  }

  protected float getMaxZoomLevel(HttpServletRequest request,
                                  IlvManagerView view)
  {
    return 30;
  }
}
```

The `init` method does not create any `XmlGrapher` object anymore. Instead, we overwrite the `prepareSession` method that has a default empty implementation to get the HTTP session. If this is a new session, we create an `XmlGrapher` object and store it as a parameter of the session. The `getManagerView` method now returns the `XmlGrapher` object stored in the session.

Multithreading Issues

The `IlvManagerServlet` class does not implement the `SingleThreadModel` interface from the Servlet API, so you can create servlets that are using the multi- or single-thread model.

If your servlet implements the SingleThreadModel interface, then you do not have to deal with concurrent access to your servlet. The servlet will be thread safe. However, this interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet.

If your servlet does not implement the SingleThreadModel interface, then you might have to be concerned with concurrent access to the servlet. All basic operations done by the IlvManagerServlet on the IlvManagerView are already synchronized. This means that you will have to take care of concurrent access only if you are doing additional actions on the IlvManagerView. In this case you can define a locking object and use the getLock method of the IlvManagerServlet. Each request handling is implemented in the following way:

```
 ... reads the request parameters ...

 synchronized(getLock(request)) {
   IlvManagerView view = getManagerView(request);

   ... handle the request ...
 }
```

By default, the getLock method returns a new object each time. This means that the section is not synchronized.

The IlvManagerServletSupport Class

The IlvManagerServlet class used in the XML Grapher example gives an easy way to create a servlet that supports the ILOG JViews thin-client protocol. Using the IlvManagerServlet class is an easy way to create a servlet but has one main drawback. You cannot add the support for the ILOG JViews thin-client protocol to an existing servlet since the IlvManagerServlet class derives from the HttpServlet class. The IlvManagerServletSupport class will allow you to do this. This class has the same API as the IlvManagerServlet but is not a servlet (that is, it does not derive from the HttpServlet class). You can thus create your own servlet and an instance of the IlvManagerServlet support class in this servlet to handle the requests coming from the ILOG JViews client side.

In our XML Grapher example, the code of the servlet can be rewritten using the IlvManagerServletSupport class as follows:

```
package demo.xmlgrapher.servlet;

import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;
import java.io.*;
import ilog.views.*;
import ilog.views.servlet.*;

import demo.xmlgrapher.*;

public class XmlGrapherServlet extends HttpServlet
{
  IlvManagerServletSupport servletSupport ;

  class MySupport extends IlvManagerServletSupport {

    private XmlGrapher xmlGrapher;

    public MySupport(ServletConfig config) {
      super();
      xmlGrapher = new XmlGrapher();

      String xmlfile = config.getInitParameter("xmlfile");

      if (xmlfile == null)
        xmlfile = config.getServletContext().getRealPath("/data/world.xml");

      try {
        xmlGrapher.setNetwork(new URL("file:" + xmlfile));
      } catch (MalformedURLException ex) {
      }

      setVerbose(true);
    }

    public IlvManagerView getManagerView(HttpServletRequest request)
      throws ServletException {
        return xmlGrapher;
      }

    protected float getMaxZoomLevel(HttpServletRequest request,
                                    IlvManagerView view) {
      return 30;
    }
  }

  /**
   * Initializes the servlet.
   */
  public void init(ServletConfig config) throws ServletException {
    servletSupport = new MySupport(config);
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    if (!servletSupport.handleRequest(request, response))
```

```
      throw new ServletException("unknow request type");
  }

  public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
       throws IOException, ServletException {
    doGet(request, response);
  }

}
```

In this code we have created a new servlet class, `XmlGrapherServlet`, that is deriving directly from the `HttpServlet` class. The `doGet` method passes the requests to an instance of the `IlvManagerServletSupport` class.

11

Scalable Vector Graphics and ILOG JViews

The ILOG JViews class library provides the ability to load Scalable Vector Graphics (SVG) files into an ILOG JViews `IlvManager`. Conversely, the contents of an `IlvManager` can easily be translated into an SVG document and saved to a file. These features allow you to interoperate with other SVG software such as SVG viewers or generators. When applicable, in the thin-client context you can also replace bitmap generation by SVG generation to gain time and interactivity.

This chapter describes what SVG is and how it can be used. We will see how to configure ILOG JViews to be able to read and write SVG files; and how to use the SVG thin-client features of ILOG JViews. And finally, we will have an in-depth explanation showing how to personalize ILOG JViews SVG features for your context and to translate your own graphic objects. The topics are:

- *Scalable Vector Graphics*
- *Loading and Saving SVG Files with ILOG JViews*
- *Using JViews SVG Thin Clients*
- *SVG Advanced Personalization*

Scalable Vector Graphics

SVG is a two-dimensional structure graphics format defined by the World Wide Web Consortium (W3C). The format is based on the eXtensible Markup Language (XML) which gives it a great interoperability.

What is in an SVG File

An SVG file describes a set of two-dimensional graphics. The following are examples of such graphics that could be found in an SVG file:

- Images through the ‘image’ element.
- Rectangles through the ‘rect’ element.
- Circles and ellipses through the ‘circle’ and ‘ellipse’ elements.
- Lines through the ‘line’ and ‘polyline’ elements.
- Polygons through the ‘polygon’ element.
- Arbitrary paths (curves, arcs, lines, and so forth) through the ‘path’ element.
- Groups of other graphic elements through the ‘g’ element.
- Text through the ‘text’ element.

These elements can be styled using XML presentation attributes on them or cascading style sheets (CSS) linked to the elements.

SVG File Example

A typical example of an SVG file would be:

```
<svg width=”640” height=”480”>
  <defs>
    <!-- the style on path elements and element with id “myid” -->
    <!-- is defined through a style sheet -->
    <style type=”text/css”>
      path {stroke-width:3;stroke:blue;fill:none}
      .dash {stroke-dasharray:5 2}
      #myid {fill:rgb(205,5,5);fill-opacity:0.5}
    </style>
    <!-- style can be complex such as a gradient... -->
    <linearGradient id=”grad” x1=”0%” y1=”0%” x2=”100%”
         y2=”100%”>
       <stop offset=”0” stop-color=”yellow”/>
       <stop offset=”0.2” stop-color=”green”/>
       <stop offset=”1” stop-color=”red”/>
     </linearGradient>
  </defs>
  <!-- the style on the rectangle is defined through XML -->
  <!-- attributes                                        -->
  <rect x=”0” y=”0” width=”100%” height=”100%”
        fill=”url(#grad)”/>
  <!-- paths use a particular syntax to defined their shape -->
  <path d=”M0 0L640 480”/>
  <path class=”dash” d=”M640 0L0 480”/>
  <!-- the style on the ellipse is defined through an inline -->
  <!-- style sheet                                           -->
  <ellipse cx=”320” cy=”240” rx=”40” ry=”30”
       style=”fill:rgb(180,10,10)”/>
  <circle id=”myid” cx=”320” cy=”240” r=”50”/>
</svg>
```

This SVG file will be rendered as a 640 x 480 rectangle filled with a linear green, yellow, and red gradient. On top on this gradient there will be two lines of thickness 3, and one of them will be a dashed line. There are also an ellipse and a circle; the color on the circle is semi-transparent (`fill-opacity:0.5`) and lets you see the color of the ellipse underneath.

For More Information

To better understand SVG and its possibilities, have a look at the SVG specification at the following URL: `http://www.w3.org/TR/SVG`. You will see that SVG provides many more features than those introduced here, such as transformations on graphic elements, filter effects, in-line animation, and scripting capabilities.

Uses for SVG Files in ILOG JViews

SVG can be used as an exchange format for ILOG JViews with various third party software that supports it. You will then be able to import data from this software to the ILOG JViews library. Conversely, you will be able to load SVG files generated by ILOG JViews into third party software such as the Batik SVG Browser (Squiggle) from Apache, which you can download free at `http://xml.apache.org/batik`.

You can also think of SVG as a means to replace bitmaps when generating data from a server and displaying them on a Web browser. In this case, the browser must be able to display an SVG document. For the moment, the easiest solution is to use the Adobe SVG Plug-in, which can be downloaded free at `http://www.adobe.com/svg`. You can find an example of such an SVG thin-client application in *Using JViews SVG Thin Clients*.

Loading and Saving SVG Files with ILOG JViews

In order to read and write SVG files, the ILOG JViews library defines a new instance of `IlvStreamFactory` specialized for SVG. We will see how to set the `ilog.views.svg.SVGStreamFactory` on the `IlvManager` and how to use it.

Configuring the SVG Factory

You should first create the manager you will work on and the factory:

```
import ilog.views.*;
import ilog.views.svg.*;

IlvManager manager = new IlvManager();
SVGStreamFactory factory = new SVGStreamFactory();
```

You will then be able to toggle some options on the factory (on or off) to change the way ILOG JViews loads and saves SVG files: how the file size should be reduced, whether the reader should ignore some data, and other options).

Here is an example:

```
// When reading SVG, the parsing of the CSS style will include
// the definitions contained in the user.css file.
factory.getReaderConfigurator().setUserStyleSheetURL(“user.css”);
// When writing SVG, the following compaction techniques will be used:
//   - style on graphic element will be factored
//   - an algorithm will be applied to polyline to remove some points
factory.getBuilderConfigurator().
    setCompactMode(SVGStreamFactory.COMPACT_STYLE |
                   SVGStreamFactory.COMPACT_POLY);
// With the option set to true on the reader configurator, when reading SVG,
// non-processible elements will be memorized
// and with the option set to true on the builder configurator, this will
// allow regenerating them later.
factory.getReaderConfigurator().setFullDocumentOn(true);
factory.getBuilderConfigurator().setFullDocumentOn(true);
```

More options can be found in the *ILOG JViews Reference Manual* for the class, such as other compaction techniques (remove invisible graphic objects, and so forth) or the ability to choose between CSS or XML styling.

Loading an SVG File

Now that the factory has been set, calling the `IlvManager.read` method will load an SVG file instead of a regular ILOG JViews file. For example:

```
try {
  manager.read(“mysvgfile.svg”);
} catch (ilog.views.io.IlvReadFileException rfe) {
   System.err.println(“The SVG file is badly formatted”);
} catch (java.io.IOException ioe) {
   System.err.println(“Cannot access the SVG file”);
}
```

Saving to an SVG File

Once an `IlvManager` has been filled dynamically or by reading an SVG or an IVL file, it is possible to save the objects to an SVG file. As with reading, the operation is similar to that for ILOG JViews native format files. For example:

```
try {
  manager.write(“mysvgfilemodified.svg”);
} catch (java.io.IOException ioe) {
  System.err.println(“Cannot access the SVG file”);
}
```

Using JViews SVG Thin Clients

Besides generating SVG files, you can use the SVG generation mechanism of ILOG JViews to provide SVG thin-client deployment to your Graphics Framework applications.

The ILOG JViews SVG thin-client support, like the ILOG JViews DHTML thin-client support (see *ILOG JViews Thin-Client Web Architecture* on page 172) is based on the Java Servlet technology.

The ILOG JViews SVG Graphics Framework thin-client support contains the following:

- An abstract servlet class than can generate SVG documents from an ILOG JViews display.
- A set of SVG scripts written in ECMAScript that will be used on the client side to display and interact with the document that was created on the server side.

A JViews SVG thin-client application provides the following features:

- Main View of the server side `IlvManager` displayed in SVG with some predefined behaviors: zooming, panning, tooltips on top of graphics objects, fixed-size graphics management, and the ability to use visibility filters with the possibility to load on demand layers invisible at initialization time.
- Overview providing the ability to navigate on the Main View.
- Layer View allowing you to display the layers of the Main View and to switch their visibility on or off.

This section will describe step by step an example of an ILOG JViews SVG thin-client application. This example is a simplified version of an example provided with the ILOG JViews distribution. Having a look at the example will give you additional information on how to code SVG:

```
<installdir>/demos/servlet/svg
```

The example can be viewed inside a Web Browser such as Internet Explorer or Netscape Communicator with the Adobe SVG Plug-in that can be downloaded for free at `http://www.adobe.com/svg`.

Developing the Server Side

The server side of an ILOG JViews SVG thin-client application is composed of two main parts: the ILOG JViews application itself, which can be any type of complex two-dimensional display built on top of the ILOG JViews Graphics Framework API, and a Servlet that produces SVG documents to the client.

In our example both server-side parts are written inside a single class defined in the file `SVGDynamicServlet.java`, located in:

```
<installdir>/demos/servlet/dynamic-svg/src/svg/SVGDynamicServlet.java
```

The ILOG JViews Application

In our particular case, the ILOG JViews application is very simple and all described in the `getManager(HttpServletRequest)` method. It simply consists of an ILOG JViews `IlvManager` filled with the contents of an ILOG JViews IVL file (`data/map.ivl`):

```
manager = new IlvManager();
try {
  // Read its contents from an IVL file.
  manager.read(getServletConfig().getServletContext().getResource("/data/map.ivl"));
} catch (java.io.IOException e) {
} catch (IlvReadFileException rfe) {
}
```

The ILOG JViews SVG Servlet

The servlet itself inherits from the `IlvSVGManagerServlet` class from the `ilog.views.svg.servlet` package.

```
import javax.servlet.*;
import javax.servlet.http.*;

import ilog.views.*;
import ilog.views.io.IlvReadFileException;
import ilog.views.svg.SVGDocumentBuilder;
import ilog.views.svg.SVGDocumentBuilderConfigurator;
import ilog.views.svg.servlet.IlvSVGManagerServlet;

public class SVGDynamicServlet extends IlvSVGManagerServlet
{
  private IlvManager manager = null;

  private static final SVGDocumentBuilderConfigurator CONFIGURATOR =
    new SVGDocumentBuilderConfigurator();

  static {
    CONFIGURATOR.setCompactMode(SVGDocumentBuilderConfigurator.COMPACT_LOD);
    CONFIGURATOR.setViewBox(new IlvRect(-2200, 4600, 3600, 3600));
  }

  /**
   * Creates a manager servlet.
   */
  public SVGDynamicServlet()
  {
    super(CONFIGURATOR);
  }


  public IlvManager getManager(HttpServletRequest r)
  {
    if (manager == null) {
      manager = new IlvManager();
      try {
        // Read its contents from an IVL file.
        manager.read(getServletConfig().getServletContext().getResource("/data/map.ivl"));
      } catch (java.io.IOException e) {
      } catch (IlvReadFileException rfe) {
      }

      } catch (IlvReadFileException rfe) {
      }
      manager.setVisible(manager.getManagerLayer("Rivers").getIndex(),
                         false, false);
      manager.getManagerLayer("Areas").
         addVisibilityFilter(new IlvZoomFactorVisibilityFilter(5,
            IlvZoomFactorVisibilityFilter.NO_LIMIT));
    }
    return manager;
  }
}
```

As you can see in the source file, it is very simple.

The `import` statements:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

are required to use the Java Servlet API.

The `import` statements:

```
import ilog.views.*;
import ilog.views.io.IlvReadFileException;
import ilog.views.svg.SVGDocumentBuilder;
import ilog.views.svg.SVGDocumentBuilderConfigurator;
import ilog.views.svg.servlet.IlvSVGManagerServlet;
```

are required for using the ILOG JViews and the ILOG JViews SVG thin-client support.

The `IlvSVGManagerServlet` class is an abstract Java subclass of the `HTTPServletClass` from the Java Servlet API. Our `SVGDynamicServlet` inherits from `IlvSVGManagerServlet` and defines only two methods: its constructor and the `getManager` method.

The constructor initializes the base class by providing an `SVGDocumentBuilderConfigurator` instance that will be used by ILOG JViews classes to configure the way the SVG document is generated.

```
private static final SVGDocumentBuilderConfigurator CONFIGURATOR =
  new SVGDocumentBuilderConfigurator();
// ...
public SVGDynamicServlet()
{
   super(CONFIGURATOR);
}
```

In our example the `SVGDocumentBuilderConfigurator` is configured such that invisible layers of the JViews display, at generation time, are not sent to the client to be able to load them only on demand. This is done through the following line:

```
CONFIGURATOR.setCompactMode(SVGDocumentBuilderConfigurator.COMPACT_LOD);
```

Then we ask to the configurator to generate the SVG document with a particular `viewBox` in order to see a specific part of the ILOG JViews display on the client:

```
CONFIGURATOR.setViewBox(new IlvRect(-2200, 4600, 3600, 3600));
```

The `getManager` method is the only abstract method of the `IlvSVGManagerServlet` class and should return an `IlvManager` that will be used by the generated SVG document. Here we simply return the manager object we created after adding some visibility filters on it to allow some layers to not always be generated:

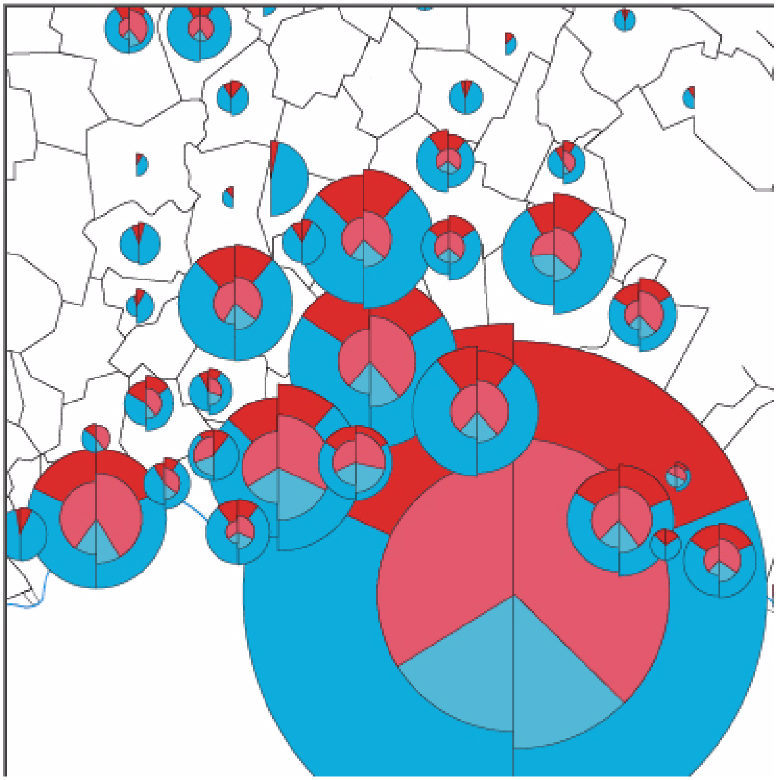
```
 manager.setVisible(manager.getManagerLayer("Rivers").getIndex(),
                        false, false);
 manager.getManagerLayer("Areas").
         addVisibilityFilter(new IlvZoomFactorVisibilityFilter(5,
            IlvZoomFactorVisibilityFilter.NO_LIMIT));
```

The first line will make the `Rivers` layer invisible on the SVG generated document. The second will make the `Areas` layer invisible if the zoom factor on the display is less than five times the initial zoom factor.

As you have seen, creating the servlet is simple. This servlet can now answer HTTP requests from a client by sending SVG documents. If you have installed the example, you can try the following HTTP request:

```
http://localhost:8080/dsvg/SVGDynamicServlet?request=image&width=400
&height=400
```

This produces the following image:



Which is the (-2200, 4600, 3600, 3600) area of the JViews manager map to a 400, 400 image.

In most cases you do not need to know the servlet parameters because the SVG scripts provided by ILOG JViews for the client side will take care of the HTTP request parameters for you.

Developing the Client Side

After creating the server side, you can create the client side. The client-side display is an SVG file that will contain several things:

- SVG elements describing the structure of the client-side display.
- JViews XML elements as metadata on the SVG elements describing how JViews client-side scripts should customize the SVG elements.
- A call to an initialization function.
- References to cascading style sheets to style the client-side display.

SVG File Example

Here is a simplified version of the `index.svg` file of the distribution example:

```
<?xml-stylesheet alternate="yes" title="default JViews style sheet"
      href="default.css" type="text/css"?>
<?xml-stylesheet alternate="no" title="style sheet" href="style.css"
      type="text/css"?>
<svg width="640" height="480"
     xmlns:ilv="http://xmlns.ilog.com/JViews/SVGToolkit"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:a3="http://ns.adobe.com/AdobeSVGViewerExtensions/3.0/"
     a3:scriptImplementation="Adobe"
     onload="JViewsSVG.Init(evt)">
  <defs>
    <!-- alternatively you could import each single file
         as in the distribution example -->
    <script xlink:href="SVGFramework.es" language="text/ecmascript"
       a3:scriptImplementation="Adobe"/>
  </defs>
  <svg id="myView" x="0%" y="0%" width="100%" height="100%">
    <metadata>
      <ilv:view type="manager" enableTooltips="true"
                xlink:href="SVGDynamicServlet"/>
    </metadata>
  </svg>
  <svg id="overview" x="-70" y="-30" width="240" height="240">
    <title>Over View</title>
    <metadata>
      <ilv:view type="over" disableZoom="true"
                enableBackground="true" xlink:href="#myView"/>
    </metadata>
  </svg>
  <svg id="legend" x="0" y="240" width="240" height="240"
      viewBox="0 0 240 240">
    <title>The View Legend</title>
    <metadata>
      <ilv:view type="layer" disableZoom="true" enableDrag="true"
                showTitle="true" enableBackground=”true” xlink:href="#myView"/>
    </metadata>
  </svg>
</svg>
```

SVG Elements

The main `svg` element contains several SVG elements which are the following:

```
  <svg id="myView" x="0%" y="0%" width="100%" height="100%">
    <metadata>
      <ilv:view type="manager" enableTooltips="true"
                xlink:href="SVGDynamicServlet"/>
    </metadata>
  </svg>
  <svg id="overview" x="-70" y="-30" width="240" height="240">
```

```
    <title>Over View</title>
    <metadata>
      <ilv:view type="over" disableZoom="true"
                enableBackground="true" xlink:href="#myView"/>
    </metadata>
  </svg>
  <svg id="legend" x="0" y="240" width="240" height="240"
      viewBox="0 0 240 240">
    <title>The View Legend</title>
    <metadata>
      <ilv:view type="layer" disableZoom="true" enableDrag="true"
                showTitle="true" enableBackground=”true” xlink:href="#myView"/>
    </metadata>
  </svg>
```

SVG Element Attributes

Each `svg` element corresponds to a view that will be displayed on the client. The `x`, `y`, `width`, and `height` attributes on the `svg` elements correspond to the rectangular region of the SVG client area in which the view will be displayed. Each view has an SVG `metadata` element that describes through the `ilv:view` element how the JViews SVG client-side scripting will display that view.

The following attributes are allowed on the `ilv:view` element:

- `type`, the possible values are `manager` (the view will display the main manager view), `overview` (the view will display an overview) and `layer` (the view will display a view of the layers).
- `disableZoom`, if `true`, the view will remain fixed in size when the user performs a zoom operation. The default value is `false`.
- `enableBackground`, if `true`, a background will be displayed under the view. The default value is `false`.
- `enableDrag`, if `true`, the user will be able to move the view by using a drag gesture. The default is `false`.

In addition to those attributes, depending on the value of the `type` attribute, some additional attributes can be recognized. These are shown in Table 11.1

Table 11.1 *Additional SVG Element Attributes*

Attribute	Recognized with the Following Types	Meaning	Default Value
`showTitle`	Layer View	If `true`, displays, as a title of the View, the contents of the ‘title’ element child of the ‘svg’ element.	`False`
`disableClip`	Manager View	If `true`, does not clip the contents of the manager view to the bounds of the ‘svg’ element.	`False`
`xlink:href`	Layer and OverView	Provides a reference to the Manager View	None. Mandatory.
`xlink:href`	Manager View	Provides a reference to the Servlet that will provide the contents of the Manager View.	None. Optional, the contents can be inlined if needed.

The Main SVG Element

Once these `svg` elements and their metadata have been written, they have to be interpreted by the client JViews SVG scripts. This is done by calling the `JViewsSVG.Init()` method at loading time on the main `svg` element. Here is that element:

```
<svg width="640" height="480"
     xmlns:ilv="http://xmlns.ilog.com/JViews/SVGToolkit"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:a3="http://ns.adobe.com/AdobeSVGViewerExtensions/3.0/"
     a3:scriptImplementation="Adobe"
     onload="JViewsSVG.Init(evt)">
  <defs>
    <!-- alternatively you could import each single file... -->
    <script xlink:href="SVGFramework.es" language="text/ecmascript"
       a3:scriptImplementation="Adobe"/>
  </defs>
</svg>
```

Main SVG Element Attributes

The main `svg` element contains several attributes:

- The `width` and `height` attributes correspond to the size the SVG document will take in the Web Browser. This can be either absolute values or percentage values. Using a percentage value, the size of the image will be rescaled when the browser size is resized.

- The `xmlns:ilv` attribute allows you to import the JViews namespace to use JViews XML elements and attributes.
- The `xmlns:xlink` attribute allows you to import the XLink namespace to use XLink.
- The `xmlns:a3` attribute allows you to import the Adobe SVG Viewer namespace to use its proprietary functionalities.
- The onload attribute references the `JViewsSVG.Init(evt)` method to allow the JViews SVG scripts to run on the client.

In addition, the main `svg` element has, as a child of the `defs` element, a reference to the scripts code. This can either reference the concatenated version (`SVGFramework.es`) or the single files needed by your application; the scripting files can be found in `<installdir>/classes/thinclient/svg`. Table 11.2 shows a summary of the files you need depending on the SVG thin-client features you are using:

Table 11.2 *Files Needed for SVG Thin Client*

Feature	ECMAScript Files Needed
Any	SVGUtil.es, SVGAbstractView.es
Main View	SVGTooltipManager.es, SVGLayer.es, SVGView.es
Over View	SVGOverview.es
Layer View	SVGTitledView.es, SVGCheckBox.es, SVGLayerView.es

Finally, the first two lines of the SVG file are:

```
<?xml-stylesheet title="default JViews style sheet"
      href="default.css" type="text/css"?>
<?xml-stylesheet title="style sheet" href="style.css"
      type="text/css"?>
```

They are processing instructions referencing cascading style sheets to give its look to the client-side display. The first referenced style sheet is the default one; it is mandatory to put that processing instruction in your SVG file. Once this is done, you can add your own style sheet (`style.css` in this case) to customize the default style; the second one will be cascaded with the first one according to CSS rules.

The style.css File

The contents of the `style.css` file are the following:

```
#legend > .backgroundRect {stroke:red;fill-opacity:0.6}
#overview > .backgroundRect {fill-opacity:1}
.overRect {fill:yellow;fill-opacity:0.8}
.title {fill-opacity:0.8}
```

This means that:

- The background rectangle (the SVG elements with the `backgroundRect` CSS class) of elements that are children of the SVG element with the `legend` ID have to be stroked with the red color and filled with an opacity of 60%.
- The background rectangle of elements that are children of the SVG element with the `overview` id have to be filled with full opacity.
- The overview rectangle (the SVG elements with the `overRect` CSS class) have to be filled with the yellow color and an 80% opacity.
- The title areas (the SVG elements with the title CSS class) have to be filled with an 80% opacity.

Here are the CSS classes that are recognized by the JViews SVG thin client and the display element to which they correspond:

Table 11.3 *CSS Classes and Display Elements*

CSS Class	Corresponds to
`backgroundRect`	Background Rectangle of a View (main view, overview, layer view).
`overRect`	OverView Rectangle.
`tooltip`	Tooltip Rectangle on a JViews Graphics Object.
`tooltipText`	Tooltip Text on a JViews Graphics Object.
`title`	Title Rectangle of a View.
`titleText`	Title Text of a View.
`enableCheckBox`	A Check Box in the enabled state.
`disableCheckBox`	A Check Box in the disabled state.

For more information about CSS in general, see the CSS specification at `http://www.w3.org/TR/REC-CSS2` and, more specifically for CSS inside SVG, see the SVG 1.0 specification at `http://www.w3.org/TR/SVG`.

Once this SVG file is written and put on the server with the `SVGDynamicServlet`, the user on the client will be able to visualize the contents of the manager on the client and to interact with it by zooming, panning, displaying the tooltips on the JViews Graphics Objects, or changing layer visibilities.

By extending the servlet, the JViews SVG generator, or the client-side SVG file, you can add your own features to the SVG thin client.

SVG Advanced Personalization

By default, if you have defined your own graphic objects, the ILOG JViews SVG generator will use a generic translator to convert them to SVG (`GenericGraphicTranslator`). However, even if it is not mandatory, you may want to define the way they are translated to SVG for several reasons (such as custom interactions); for this you will need to write some additional code. This code may also apply when you want to redefine the way a standard ILOG JViews `IlvGraphic` will be translated.

Customizing a Graphic Object: an Example

The following three-part example illustrates how you can customize the generation of a graphic object:

- *Subclassing a Graphic Object*
- *Creating a Translator and Setting It on the Builder Configurator*
- *Customizing the SVG Factory*

Subclassing a Graphic Object

Assume your subclassed graphic object draws a blue Bezier curve into a rectangle from the left bottom point to the right top point:

```
package mypackage;

import java.awt.*;
import ilog.views.*;
import ilog.views.graphic.*;

public class MyGraphic extends IlvRectangle
{
  private IlvPoint[] pts = new IlvPoint[4];
  private float[] dash = {4, 2};

  public MyGraphic(IlvRect rect)
  {
     super(rect);
     for (int i = 0; i < 4; i++)
        pts[i] = new IlvPoint();
     computeBezier();
  }

  private computeBezier()
  {
    pts[0].x = drawrect.x;
    pts[0].y = drawrect.y + drawrect.height;
    pts[1].x = drawrect.x + drawrect.width / 4;
    pts[1].y = drawrect.y + drawrect.height / 4;
    pts[2].x = drawrect.x + 3*drawrect.width / 4;
    pts[2].y = drawrect.y + 3*drawrect.height / 4;
    pts[3].x = drawrect.x + drawrect.width;
    pts[3].y = drawrect.y;
  }

  public void draw(Graphics g, IlvTransformer t)
  {
     super.draw(g, t); // will draw the rectangle
     // Will draw a Bezier in blue
     g.setColor(Color.blue);
     IlvGraphicUtil.DrawBezier(g, pts, 4, 1,
                               IlvStroke.JOIN_MITER, IlvStroke.CAP_ROUND,
                               dash, t);
  }

  public void applyTransform(IlvTransform t)
  {
     super.applyTransform(t);
     computeBezier();
  }

  // ...
}
```

Creating a Translator and Setting It on the Builder Configurator

If you do not want your object to use the generic translation mechanism, you should first implement the `SVGDocumentBuilderConfigurator.GraphicTranslator` interface to be able to translate `MyGraphic` instances to SVG `Element` instances. The `translate` method of your class should build a new `Element` instance, and after filling it with the information contained in the graphic object and applying a style, the method should return that element. To fill the `Element`, you can use regular DOM methods (`Element.setAttribute()`) or the methods provided by the SVG DOM. Note, however, that only limited support of SVG DOM is implemented in ILOG JViews, and not all the methods are accessible.

```
import org.w3c.dom.*;
import org.w3c.svg.dom.*;

public class MyGraphicTranslator
       implements SVGDocumentBuilderConfigurator.GraphicTranslator
{
  // Method that translates the graphic to SVG
  public Element translate(IlvGraphic graphic, IlvTransformer t,
                           SVGDocumentBuilder builder)
  {
    SVGDocument doc = builder.getDocument();

    // The SVG ’group’ element will contain
    // all drawings necessary to display MyGraphic.
    SVGGElement group = (SVGGElement)doc.createElementNS
```

```
                                        ("http://www.w3.org/2000/svg", "g");

    // We first add to the group the element corresponding
    // to the parent class of MyRect (IlvRectangle).
    group.appendChild(builder.getConfigurator().
       getTranslator(“ilog.views.graphic.IlvRectangle”).
       translate(graphic, t));

    // Then we create the SVG element corresponding to
    // the drawing added at MyGraphic level.
    SVGPathElement path = (SVGPathElement)doc.createElementNS
                                        ("http://www.w3.org/2000/svg", "path");
    IlvRect rect = graphic.boundingBox(graphic, t);
    SVGList list = path.getPathSegList();
    // Go to the beginning point of the Bezier.
    list.appendItem(path.createPathSegMovetoAbs(rect.x,
                                                rect.y + rect.height));
// Add the Bezier.
    list.appendItem(path.
      createPathSegCurvetoCubicAbs(rect.x + rect.width,
                                   rect.y,
                                   rect.x + rect.width / 4,
                                   rect.y + rect.height / 4,
                                   rect.x + 3*rect.width / 4,
                                   rect.y + 3*rect.height / 4));

    // Start to style the path.
    builder.startStyleElement(path, null);
    // Really apply the style (blue color...).
    builder.appendStyle("stroke", "blue");
    builder.appendStyle("stroke-width", "1");
    builder.appendStyle("stroke-join", "miter");
    builder.appendStyle("stroke-cap", "round");
    builder.appendStyle("stroke-dasharray", "4 2");
    // Finish to style the path.
    builder.endStylingElement();
    group.appendChild(path);

    return group;
  }
}
```

Customizing the SVG Factory

Now that the new translator is defined, you should create a well-configured builder configurator. If you are using an `SVGOutputStream` you can do the following:

```
SVGStreamFactory factory = new SVGStreamFactory();
factory.getBuilderConfigurator().putTranslator("mypackage.MyGraphic", new
 MyGraphicTranslator());
```

Of course, you should not forget to set this new factory on your manager instead of the regular one.

If you are in SVG thin-client context, you just have to call putTranslator method on the SVGBuilderConfigurator instance you used for your Servlet.

Customizing the SVG DOM Generated by the SVG Thin Client

In the SVG thin-client context, in addition to defining your own way to export IlvGraphic instances to SVG, you may want to modify the SVG DOM instance that will be sent to the client beforehand. For this you just need to redefine the generateSVGDocument method of the IlvSVGManagerServlet class. For example, if you need to put an SVG drop-shadow effect arrow on a particular object, you can do the following:

```
protected Document generateSVGDocument(HttpServletRequest request,
                                       int width, int height,
                                       String[] requestedLayers)
   throws ServletException
  {
    Document document = super.generateSVGDocument(request,
                                                  width, height,
                                                  requestedLayers);
    // create the drop-shadow filter effect
    Element filter = document.createElementNS(null, "filter");
    filter.setAttribute("id", "drop");
    filter.setAttribute("filterUnits", "objectBoundingBox");
    filter.setAttribute("x", "-0.1");
    filter.setAttribute("y", "-0.1");
    filter.setAttribute("width", "1.2");
    filter.setAttribute("height", "1.2");
    Element blur = document.createElementNS(null, "feGaussianBlur");
    blur.setAttribute("in", "SourceAlpha");
    blur.setAttribute("stdDeviation", "2");
    blur.setAttribute("result", "balpha");
    Element offset = document.createElementNS(null, "feOffset");
    offset.setAttribute("in", "balpha");
    offset.setAttribute("dx", "4");
    offset.setAttribute("dy", "4");
    offset.setAttribute("result", "oba");
    Element merge = document.createElementNS(null, "feMerge");
    Element node = document.createElementNS(null, "feMergeNode");
    node.setAttribute("in", "oba");
    merge.appendChild(node);
    node = document.createElementNS(null, "feMergeNode");
    node.setAttribute("in", "SourceGraphic");
    merge.appendChild(node);
    filter.appendChild(blur);
    filter.appendChild(offset);
    filter.appendChild(merge);
    // add the drop-shadow filter effect to the document
    document.getDocumentElement().appendChild(filter);
    // set the effect on the "myGraphic" element if it is in one
    // of the generated layers:
    Element elt = document.getElementById("myGraphic");
    if (elt != null)
      elt.setAttribute("filter", "url(#drop)");
    // return the modified document
    return document;
  }
```

SVG Features Supported when Reading an SVG File

To help you build SVG files that will be fully understood by the SVG reader of ILOG JViews, use the following tables for supported/unsupported SVG elements and CSS properties.

Table 11.4 *Supported SVG Elements*

Element Name	Attributes Not Supported on this Element
a	xlink:role, xlink:acrole, xlink:actuate
circle	
clipPath	clipPathUnits
defs	
desc	
ellipse	
g	
image	
line	
linearGradient	
metadata	
path	
pattern	patternUnits, patternTransform
polygon	
polyline	
radialGradient	
rect	
stop	
style	!important rules are not supported

Table 11.4 Supported SVG Elements (Continued)

Element Name	Attributes Not Supported on this Element
svg	zoomAndPan
switch	
symbol	refX, refY, viewBox, preserveAspectRatio
text	textLength, lengthAdjust
textPath	textLength, lengthAdjust, startOffset, method, spacing
title	
tspan	multiple values x, y, dx, dy (single values supported), rotate, textLength
use	

Table 11.5 Supported CSS Properties

Property Name	Remark
clip-path	URI local to the file only.
color	
color-interpolation	Supported on linearGradient and radialGradient.
fill	URI local to the file only, ICC colors are not supported.
fill-opacity	
fill-rule	
font-family	
font-size	Relative identifiers are not supported.
font-stretch	Relative identifiers are not supported.
font-style	
font-weight	Relative identifiers are not supported.
stop-color	ICC colors are not supported.
stop-opacity	

***Table 11.5** Supported CSS Properties (Continued)*

Property Name	Remark
`stroke`	URI local to the file only, ICC colors are not supported.
`stroke-dasharray`	
`stroke-dashoffset`	
`stroke-linecap`	
`stroke-linejoin`	
`stroke-miterlimit`	
`stroke-opacity`	
`stroke-width`	
`text-anchor`	
`visibility`	

APPENDIX

A

Differences Between ILOG Views and ILOG JViews

The basic concepts of ILOG JViews are taken from the ILOG Views C++ library. To help you get started with ILOG JViews, this section points out the most important differences between ILOG Views C++ and ILOG JViews. You can skip this section if you are not an ILOG Views C++ developer. The following topics are discussed:

- *Main Concepts*
- *Drawing Operations*
- *IlvManagerViewHook*
- *Miscellaneous*

Main Concepts

You may have noticed that the main concepts are the same even if some classes have changed name.

Here are the main classes of ILOG JViews and their corresponding classes in ILOG Views:

- `IlvManager`, in charge of organizing graphic objects in several layers and displaying them in several views, corresponds to the ILOG Views `IlvManager` class.

- IlvGrapher, which manages nodes and links, corresponds to the ILOG Views IlvGrapher class.
- IlvManagerViewInteractor, the base class for interaction in a view, corresponds to the ILOG Views IlvManagerViewInteractor class.
- IlvGraphic, the base class for graphic objects, corresponds to the IlvGraphic class.
- IlvObjectInteractor, corresponds to the ILOG Views IlvManagerObjectInteractor class. This class was renamed since it can be used outside of an IlvManager.
- IlvManagerView, view displaying manager contents. In ILOG Views, any subclass of IlvView can be used to display the contents of a manager.

Some classes in ILOG Views are not available in ILOG JViews. Some of these classes may have a corresponding class in ILOG JViews or in AWT.

- IlvContainer is not available in ILOG JViews and can be replaced by an IlvManager.
- IlvViewObjectInteractor of ILOG Views is not available since there is no IlvContainer. It can be replaced by the IlvObjectInteractor class of ILOG JViews.
- IlvSCManagerRectangle, which displays scroll bars around the view of a manager can be replaced by the class IlvScrollManagerView of ILOG JViews.
- IlvGadget of ILOG Views does not exist. ILOG JViews is mainly a library to manipulate 2D graphic objects. The portable standard GUI components of ILOG Views, such as buttons, text fields, and string lists, are not part of ILOG JViews because they are already available for Java developers in the AWT or Swing libraries. Therefore, you will not find the class IlvGadget and its subclasses in ILOG JViews.
- No gauge classes are available in ILOG JViews.

Drawing Operations

All drawing operations in ILOG JViews are based on the AWT library of Java.

ILOG JViews does not provide an IlvDisplay class. This class, which groups the management of display resources, is not needed because similar mechanisms are available in AWT.

The class IlvPalette is not available in ILOG JViews either. It is replaced by storing references to the resources (Font or Color) needed in the graphic object. Some graphic resources, such as line style and thickness, or fill style of an IlvPalette object, cannot be translated in ILOG JViews. These resources are not available in AWT.

Here are the differences between the `draw` method of ILOG Views and ILOG JViews (both of the class `IlvGraphic`):

- ILOG Views signature:

```
virtual void draw(IlvPort* dst, const IlvTransformer* t = 0,
                  const IlvRegion* clip = 0) const;
```

- ILOG JViews signature:

```
public void draw(Graphics dst, IlvTransformer t)
```

The `IlvPort` parameter of ILOG Views is replaced by an AWT `Graphics` object. The class `java.awt.Graphics` is a graphic context that allows drawing on a view. It manages the clipping area, the current drawing color and font, and the logical pixel operation (XOR, Paint).

One of the main differences between ILOG Views and ILOG JViews is the coordinate system. In ILOG JViews, the coordinate system uses floating-point values (see classes `IlvRect` and `IlvPoint`), whereas in ILOG Views, the coordinate system is in integer values. This means that there is some additional work to do when you override the `draw` method of a graphic object. The coordinates must be translated into integer values to be able to draw on a view. This is done by using the following methods:

`IlvTransformer.applyFloor`

`IlvRect.floor`

`IlvPoint.floor`

These methods use the `Math.floor` method to do the conversion from `float` to `int`.

IlvManagerViewHook

The ILOG Views class `IlvManagerViewHook` detects events such as the modifications made to the contents of a manager. In ILOG JViews, this functionality has been split into several listeners:

- A listener for the changes made to the contents of the manager: `ManagerContentChangeListener`
- A listener for the modifications made to the transformer of a view: `TransformerListener`.
- A listener for the modifications made to the interactor of a view: `InteractorListener`.

The `beforeDraw` and `afterDraw` methods of the hook can be implemented by overriding the `paint` method of the class `IlvManagerView`.

Miscellaneous

The class IlvTimer of ILOG Views can be replaced by a new Thread class in Java.

The class IlvManagerGrid was renamed IlvGrid because it can be used in a context where there is no manager.

The classes IlvManagerMakeStringInteractor and IlvManagerMakeTextInteractor are replaced by an interactor named IlvEditLabelInteractor that allows you to create and edit any type of labeled object.

Glossary

accelerator

In ILOG JViews, an immediate command triggered by a specific keyboard event. The event is recognized by the manager, which calls a predefined function.

affine transformation

A composite linear remapping of an image's coordinate geometry to correct for perspective distortions.

bounding box

The smallest possible horizontal rectangle surrounding one or more objects.

double buffering

A technique for displaying numerous graphics on a screen without flicker.

grapher

A high-level ILOG JViews functionality allowing you to create programs that both include and represent hierarchical information with graphic objects. A grapher is a type of manager, an instance of the `IlvGrapher` class, a subclass of the `IlvManager` class.

grapher pin

In ILOG JViews, a fixed connection point for a link on a node in a grapher.

graphic bag

An interface that describes the methods to be implemented by a class that contains several graphic objects. An example of a graphic bag is the class `IlvManager`, which manages a large number of graphic objects.

graphic object

In ILOG JViews, a defined pictorial entity having functionality which allows it to be drawn, saved, and reshaped. There are many predefined graphic objects, such as rectangles and ellipses, as well as more complex objects, such as shaded labels and zoomable icons.

handle object

In ILOG JViews, an object used to reference another object. Using handle objects, the same object can be displayed more than once without being duplicated.

hit testing Mapping from a given display point to an object to determine if the point is part of the object space.

interactor *See* object interactor, view interactor.

layer Storage area of manager in which graphic objects are placed. A manager uses multiple layers, referenced by index numbers. Objects in a higher-numbered layer are displayed in front of objects of a lower-numbered layer.

link The visual representation of connections between nodes. Links are also graphic objects, instances of the `IlvLinkImage` class or its subclasses.

link connector In ILOG JViews, the system that allows you to control the connections of links to nodes in a grapher.

listener In ILOG JViews, an interface allowing you to connect events that occur in the manager view with actions to be performed.

manager In ILOG JViews, a container for grouping graphic objects and for coordinating their behavior and display in multiple layers and views.

manager view The AWT component where the graphic objects of a manager are displayed. To display graphic objects contained in the different layers of a manager, you create at least one view, and often multiple views. The manager lets you connect as many views as you require to display graphic objects.

node A visual reference point in a hierarchy of information. A node is a graphic object, as it is an instance of a subclass of the `IlvGraphic` class.

object interactor In ILOG JViews, a behavior that can be attached to an object to make it work in a certain way. For example, a button interactor attached to an object makes that object act like a button.

pin *See* grapher pin.

quadtree A structure composed of four quadrants which recursively subdivide themselves by four, allowing you to achieve manageable datasets. This permits quick access to information, as the quadtree can confine its search to quadrants covering a particular area of interest.

selection object A graphic object created by ILOG JViews to display a selected object within a manager. These objects are stored in the manager. Unlike regular graphic objects, they are internally managed and cannot be manipulated.

transformer A 2D affine transformation that performs a linear mapping from 2D coordinates to other 2D coordinates. A transformer can be a scale, a translation, or a rotation.

triple buffering A technique for increasing the speed of an application by minimizing redrawing of static components.

vector graphic A digital image made up of geometric graphic elements such as points, arcs, and lines that are defined by mathematical formulas. This mathematical formulation allows for the size of the graphic to be changed without a loss in its image quality.

view *See* manager view.

view interactor A behavior that can be applied to a view as a whole, affecting all the objects in the view.

See also object interactor.

Index